

Python

Python Data Structures Repo & Website

Blog



index

<https://bgoonz42.gitbook.io/python/>



<https://bootcamp42.gitbook.io/python/>

<https://bootcamp42.gitbook.io/python/>

My Website:



Python Notes

<https://ds-unit-5-lambda.netlify.app/>

My Notion



<https://golden-lobe-519.notion.site/Data-Structures-c3fe3debbe494b929ed2f20070b631f8>

<https://golden-lobe-519.notion.site/Data-Structures-c3fe3debbe494b929ed2f20070b631f8>



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ds-algo-forked-iezyk?file=/index.html>



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ds-algo-forked-qedq8>

Python syntax was made for readability, and easy editing. For example, the python language uses a `:` and indented code, while javascript and others generally use `{}` and indented code.

Lets create a python 3 repl, and call it *Hello World*. Now you have a blank file called *main.py*.

Now let us write our first line of code:

helloworld.py

```
print('Hello world!')
```

Brian Kernighan actually wrote the first “Hello, World!” program as part of the documentation for the BCPL programming language developed by Martin Richards.

Now, press the run button, which obviously runs the code. If you are not using replit, this will not work. You should research how to run a file with your text editor.

If you look to your left at the console where hello world was just printed, you can see a `>`, `>>>`, or `$` depending on what you are using. After the prompt, try typing a line of code.

```
1 Python 3.6.1 (default, Jun 21 2017, 18:48:35)
2 [GCC 4.9.2] on linux
3 Type "help", "copyright", "credits" or "license" for more information.
4 > print('Testing command line')
5 Testing command line
6 > print('Are you sure this works?')
7 Are you sure this works?
8 >
```

The command line allows you to execute single lines of code at a time. It is often used when trying out a new function or method in the language.

Another cool thing that you can generally do with all languages, are comments. In python, a comment starts with a `#`. The computer ignores all text starting after the `#`.

shortcom.py

```
# Write some comments!
```

If you have a huge comment, do **not** comment all the 350 lines, just put `'''` before it, and `'''` at the end. Technically, this is not a comment but a string, but the computer still ignores it, so we will use it.

longcom.py

```
1  '''
2  Dear PYer,
3  I am confused about how you said you could use triple quotes to make
4  SUPER
5  LONG
6  COMMENTS
7  !
8
9  I am wondering if this is true,
10 and if so,
11 I am wondering if this is correct.
12
13 Could you help me with this?
14
15 Thanks,
16 Random guy who used your tutorial.
17 '''
18 print('Testing')
```

Unlike many other languages, there is no `var`, `let`, or `const` to declare a variable in python. You simply go `name = 'value'`.

vars1.py

```
1 x = 5
2 y = 7
3 z = x*y # 35
4 print(z) # => 35
```

Remember, there is a difference between integers and strings. *Remember: String = ""*. To convert between these two, you can put an int in a `str()` function, and a string in a `int()` function. There is also a less used one, called a float. Mainly, these are integers with decimals. Change them using the `float()` command.

vars2.py

```
1 x = 5
2 x = str(x)
3 b = '5'
4 b = int(b)
5 print('x = ', x, '; b = ', str(b), ';') # => x = 5; b = 5;
```

Instead of using the `,` in the print function, you can put a `+` to combine the variables and string.

There are many operators in python:

- `+`
- `-`
- `/`
- `*` These operators are the same in most languages, and allow for addition, subtraction, division, and multiplication. Now, we can look at a few more complicated ones:
- `%`
- `//`
- `**`
- `+=`
- `-=`

- `/=`
- `*=` Research these if you want to find out more...

simpleops.py

```
1 x = 4
2 a = x + 1
3 a = x - 1
4 a = x * 2
5 a = x / 2
```

You should already know everything shown above, as it is similar to other languages. If you continue down, you will see more complicated ones.

complexop.py

```
1 a += 1
2 a -= 1
3 a *= 2
4 a /= 2
```

The ones above are to edit the current value of the variable.

Sorry to JS users, as there is no `i++;` or anything.

Fun Fact:

The python language was named after Monty Python.

If you really want to know about the others, view Py Operators

Like the title?

Anyways, a `'` and a `"` both indicate a string, but **do not combine them!**

quotes.py

```
1 x = 'hello' # Good
2 x = "hello" # Good
```

```
3 x = "hello' # ERRORRR!!!
```

slicing.py

String Slicing

You can look at only certain parts of the string by slicing it, using `[num:num]`.

The first number stands for how far in you go from the front, and the second stands for how far in you go from the back.

```
1 x = 'Hello everybody!'
2 x[1] # 'e'
3 x[-1] # '!'
4 x[5] # ' '
5 x[1:] # 'ello everybody!'
6 x[:-1] # 'Hello everybod'
7 x[2:-3] # 'llo everyb'
```

Methods and Functions

Here is a list of functions/methods we will go over:

- `.strip()`
- `len()`
- `.lower()`
- `.upper()`
- `.replace()`
- `.split()`

I will make you try these out yourself. See if you can figure out how they work.

strings.py

```
1 x = " Testing, testing, testing, testing      "
2 print(x.strip())
3 print(len(x))
4 print(x.lower())
```

```
5 print(x.upper())
6 print(x.replace('test', 'runn'))
7 print(x.split(','))
```

Good luck, see you when you come back!

Input is a function that gathers input entered from the user in the command line. It takes one optional parameter, which is the users prompt.

inp.py

```
1 print('Type something: ')
2 x = input()
3 print('Here is what you said: ', x)
```

If you wanted to make it smaller, and look neater to the user, you could do...

inp2.py

```
print('Here is what you said: ', input('Type something: '))
```

Running:

inp.py

```
1 Type something:
2 Hello World
3 Here is what you said: Hello World
```

inp2.py

```
1 Type something: Hello World
2 Here is what you said: Hello World
```

Python has created a lot of functions that are located in other .py files. You need to import these **modules** to gain access to them. You may wonder why python did this. The purpose of separate modules is to make python faster. Instead of storing millions and millions of functions, it only needs a few basic ones. To import a module, you must write

`import <modulename>` . Do not add the .py extension to the file name. In this example , we will be using a python created module named random.

module.py

```
import random
```

Now, I have access to all functions in the random.py file. To access a specific function in the module, you would do `<module>.<function>` . For example:

module2.py

```
1 import random
2 print(random.randint(3,5)) # Prints a random number between 3 and 5
```

Pro Tip:

Do `from random import randint` to not have to do `random.randint()` , just `randint()`
To import all functions from a module, you could do `from random import *`

Loops allow you to repeat code over and over again. This is useful if you want to print Hi with a delay of one second 100 times.

for Loop

The for loop goes through a list of variables, making a separate variable equal one of the list every time.

Let's say we wanted to create the example above.

loop.py

```
1 from time import sleep
2 for i in range(100):
3     print('Hello')
4     sleep(.3)
```

This will print Hello with a .3 second delay 100 times. This is just one way to use it, but it is usually used like this:

loop2.py

```
1 import time
2 for number in range(100):
3     print(number)
4     time.sleep(.1)
```

while Loop

The while loop runs the code while something stays true. You would put `while <expression>`. Every time the loop runs, it evaluates if the expression is True. If it is, it runs the code, if not it continues outside of the loop. For example:

while.py

```
1 while True: # Runs forever
2     print('Hello World!')
```

Or you could do:

while2.py

```
1 import random
2 position = '<placeholder>'
3 while position != 1: # will run at least once
4     position = random.randint(1, 10)
5     print(position)
```

The if statement allows you to check if something is True. If so, it runs the code, if not, it continues on. It is kind of like a while loop, but it executes **only once**. An if statement is written:

if.py

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4     print('num is 3. Hooray!!!!')
5 if num > 5:
6     print('Num is greater than 5')
7 if num == 12:
8     print('Num is 12, which means that there is a problem with the python lan
```

Now, you may think that it would be better if you could make it print only one message. Not as many that are True. You can do that with an `elif` statement:

elif.py

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4     print('Num is three, this is the only msg you will see.')
5 elif num > 2:
6     print('Num is not three, but is greater than 1')
```

Now, you may wonder how to run code if none work. Well, there is a simple statement called `else`:

else.py

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4     print('Num is three, this is the only msg you will see.')
5 elif num > 2:
```

```
6     print('Num is not three, but is greater than 1')
7 else:
8     print('No category')
```

So far, you have only seen how to use functions other people have made. Let's use the example that you want to print a random number between 1 and 9, and print different text every time. It is quite tiring to type:

Characters: 389

nofunc.py

```
1 import random
2 print(random.randint(1, 9))
3 print('Wow that was interesting.')
4 print(random.randint(1, 9))
5 print('Look at the number above ^')
6 print(random.randint(1, 9))
7 print('All of these have been interesting numbers.')
8 print(random.randint(1, 9))
9 print("these random.randint's are getting annoying to type")
10 print(random.randint(1, 9))
11 print('Hi')
12 print(random.randint(1, 9))
13 print('j')
```

Now with functions, you can seriously lower the amount of characters:

Characters: 254

functions.py

```
1 import random
2 def r(t):
3     print(random.randint(1, 9))
4     print(t)
5 r('Wow that was interesting.')
6 r('Look at the number above ^')
7 r('All of these have been interesting numbers.')
8 r("these random.randint's are getting annoying to type")
9 r('Hi')
```

```
10 r('j')
```

Chapter 01 - Getting Ready with Python

Installing Python 3, And Launching Python Shell

This video should help you get up and running with Python 3

- Installing Python 3 and Launch Python Shell

Installing Python is really a cakewalk. Search for “Python download” on www.google.com. Download the installable and install it.

A quick word of caution on Windows

- Make sure that you have the check-box “Add Python 3.6 to PATH”, checked.

Once you have installed Python, you can launch the Python Shell.

- Windows - Launch cmd prompt by typing in ‘cmd’ command.
- Mac or Linux - Launch up terminal.

Command to launch Python 3 is different in Mac.

- In Mac, type in `python3`
- In other operating systems, including windows, type `python`

You can type code in python shell and code as well!

You can use `print(5*4)` , and it shows `20` .

You can execute the code, and the shell would immediately give you output.

Using the the Python Shell is an awesome way to learn Python.

Chapter 02 - Introduction To Python Programming

Most programmers find programming a lot of fun, and besides, it also gets their work done.

Programming mainly involves *problem solving*, where one makes use of a computer to solve a real world problem.

During our journey here, we will approach programming in a very different way. We will not only introduce you to the Python language, but also help you pick up essential problem solving skills.

As a programmer, you need to be able to look at a problem, and identify the important programming concepts relevant to solving it. Finally, you need to be able to use the language features and syntax, to express your solution on the computer. While all this looks complex, we want to make it easy for you. Together, we will tackle a variety of programming challenges, using these same steps. We will start with simple challenges (such as a Multiplication Table), and gradually increase the difficulty level over the duration of this book.

Learning to program is a lot like learning to ride a bicycle. The first few steps are the most challenging ones.

Once you get over these initial steps, your experience will become more and more enjoyable.

Are you ready for your first programming challenge? Let's get going now! We wish you all the best.

Summary

In this step, we:

- Were introduced to the concept of problem solving
- Understood how good programmers approach problem solving

Step 01: Our First Programming Challenge

Our first *programming challenge* aims to do, what every kid does in math class: read out a multiplication table. We now want to give this task to the computer. Here is the statement of our problem:

The Print Multiplication Table Challenge (PMT-Challenge)

1. Compute the multiplication table for 5 , with entries from 1 to 10 .
2. Display this table.

The display needs to be:

$$5 * 1 = 5$$

$$5 * 2 = 10$$

$$5 * 3 = 15$$

$$5 * 4 = 20$$

$$5 * 5 = 25$$

$$5 * 6 = 30$$

$$5 * 7 = 35$$

$$5 * 8 = 40$$

$$5 * 9 = 45$$

$$5 * 10 = 50$$

This is the challenge. For convenience, let's give it a label, say *PMT-Challenge*. What would be the important concepts we need to learn, to solve this challenge? The following list of concepts would be a good starting point:

- **Statements**
- **Expressions**
- **Variables**
- **Literals**
- **Conditionals**
- **Loops**
- **Methods**

In the rest of this chapter, we will introduce these concepts to you, one-by-one. We will also show you how learning each concept, takes us closer to a solution to *PMT-Challenge*.

Summary

In this step, we:

- Stated our first programming challenge
- Identified what programming concepts we need to learn, to solve this challenge

Step 02: Breaking Down *PMT-Challenge*

Typically when we do programming, we have problems. Solving the problem typically need a step-by-step approach. Common sense tells us that to solve a complex problem, we break it into smaller parts, and solve each part one by one. Here is how any good programmer worth her salt, would solve a problem:

- Simplify the problem, by breaking it into sub-problems
- Solve the sub-problems in stages (in some order), using the language
- Combine these solutions to get a final solution

The *PMT-Challenge* is no different! Now how do we break it down, and where do we really start? Once again, your common sense will reveal a solution. As a first step, we could get the computer to calculate say, $5 * 3$. The second thing we can do, is to try and print the calculated value, in a manner similar to $5 * 3 = 15$. Then, we could repeat what we just did, to print out all the entries of the 5 multiplication table. Let's put it down a little more formally:

Here is how our draft steps look like

- Calculate $5 * 3$ and print result as 15
- Print $5 * 3 = 15$ (15 is result of previous calculation)
- Do this ten times, once for each table entry (going from 1 to 10)

Let's start with that kind of a game plan, and see where it takes us.

Summary

In this step, we:

- Learned that breaking down a problem into sub-problems is a great help
- Found a way to break down the *PMT-Challenge* problem

Step 03: Introducing Operators And Expressions

Let's focus on solving the first sub-problem of *PMT-Challenge*, the numeric computation. We want the computer to calculate $5 * 5$ for example, and print 25 for us. How do we get it to

do that? That's what we would be looking at in this step.

Snippet-01: Introducing Operators

Launch up Python shell. We want to calculate `5 * 5`. How do we do that?

Using our knowledge of school math, let's try `5 X 5`.

```
1      >>> 5 X 5
2      File "< stdin >", line 1
3      5 X 5
4          ^
5      SyntaxError: invalid syntax
```

The Python Shell hits back at us, saying "*invalid syntax*". This is how Python complains, when it doesn't fully understand the code you type in. Here, it says our code has a "**SyntaxError**".

The reason why it complains, is because '`x`' is not a valid **operator** in Python.

The way you can do multiplication is by using the '`*`' *operator*.

"*5 into 5*" is achieved by the code `5 * 5`, and you can see the result `25` being printed.

Similarly, `5 * 6` gives us `30`.

```
1      >>> 5 * 6
2      30
```

There are a wide range of other operators in Python:

- `5 + 6` gives a result of `11`.
- `5 - 6` leads to `-1`.

```
1  >>> 5 + 6
2  11
3  >>> 5 - 6
4  -1
```

`10 / 2`, gives an output of `5.0`. There is one interesting operator, `**`. Let's try `10 ** 3`. We ran this code, and the result we get is `1000`. Yes you guessed right, the operator performs "to the power of". "`10` to the power of `3`" is `10 * 10 * 10`, or `1000`.

```
1      >>> 10 / 2
2      5.0
3      >>> 10 ** 3
4      1000
```

Another interesting operator is `%`, called "*modulo*", which computes the remainder on integer division. If we do `10 % 3`, what is the remainder when `10` is divided by `3`? `3 * 3` is `9`, and `10 - 9` is `1`, which is what `%` returns in this case.

Let's look at some terminology:

- Whatever pieces of code we gave Python shell to run, are called **expressions**. So, `5 * 5`, `5 * 6` and `5 - 6` are all *expressions*. An expression is composed of *operators* and **operands**.
- In the expression `5 * 6`, the two values `5` and `6` are called *operands*, and the `*` operator *operates* on them.
- The values `5` and `6` are **literals**, because those are constants which cannot be changed.

The cool thing about Python, is that you can even have expressions with multiple operators. Therefore, you can form an expression with `5 + 5 + 5`, which evaluates to `15`. This is an expression which has three *operands*, and two `+` operators. You can even have expressions with different types of operators, such as in `5 + 5 * 5`.

```
1      >>> 5 + 5 + 5
2      15
3      >>> 5 + 5 * 5
4      30
```

Try and play around with the expressions, and understand the output which results.

Summary

In this step, we:

- Learned how to give code input to the Python Shell
- Understood that Python has a predefined set of operators
- Used a few types of basic operators and their operands, to form expressions

Step 04: Programming Exercise IN-PE-01

At this stage, your smile tells us that you enjoy evaluating Python expressions. What if we tickle your mind a bit, to make sure it hasn't fallen asleep? Here is your first programming exercise.

Exercises

1. Write an expression to calculate the number of minutes in a day.
2. Write an expression to calculate the number of seconds in a day.

Note

You need to solve these problems by yourself. If you are able to work them out, that's fantastic! But if not, that's part of the learning process.

Solutions

Solution 1

```
1      >>> 24 * 60
2
3      1440
```

We wanted to calculate the number of minutes in a day. How do we do that? Think about this...

- How many number of hours are there in a day? 24 .
- And how many minutes does each hour have? It's 60 .
- So if you want to find out the number of minutes in a day, it's 24 * 60 , which is 1440 .

Solution 2

```
1      >>> 24 * 60 * 60
2
3      86400
```

How many seconds are there in a day?

- Let's start with the number of hours, `24`.
- The number of minutes in an hour is `60`, and
- The number of seconds in a minute is `60` as well.
- So it's `24 * 60 * 60`, or `86400`.

Summary

In this step, we:

- Solved a Programming Exercise involving common scenarios, using Python code involving:
 - Expressions
 - Operators
 - Literals

Step 05: Puzzles On Expressions

Let's look at a few puzzles related to expressions, in this step. Before that, let's revise some of the terminology we had learned earlier.

`5 + 6 + 10` is an example of an expression. In this expression, `5`, `6` and `10` are operands. The `+` here is the operator. You can have multiple operators in an expression. We also did mention that the operands, namely `10`, `6` and `5`, are literals. Their values will not change.

Here are a few puzzles coming up, to explore aspects of expressions.

Snippet-01: Puzzles On Expressions

Think about what would happen when you do something of this kind: `5 $ 2`. You're right, it would throw a `SyntaxError`. When Python does not understand the code you type in, it reports an error. Here, the expression we're typing is `5 $ 2`, which does not make sense to Python, hence the `SyntaxError`.

```
1      >>> 5 $ 2
2      File "< stdin >", line 1
3      5 $ 2
4      ^
5      SyntaxError: invalid syntax
6      >>> 5$2
7      File "< stdin >", line 1
8      5 $ 2
9      ^
10     SyntaxError: invalid syntax
```

Let's say we type in `5+6+10`, without any spaces between the operands, and the operators. What do you think will happen? Surprisingly, the Python Shell does calculate the value!

```
1      >>> 5+6+10
2      21
```

In an expression, using spaces makes it easier for you to read it, but it's not mandatory.

`5 + 6 + 10` is easier to read than `5+6+10`, but does not make any difference to the Python compiler.

The next puzzle tries to evaluate `5 / 2`, which is “`5` divided by `2`”. What would be the output? `2.5`.

```
1      >>> 5/2
2      2.5
```

If you're coming from other programming languages like Java or C, this might be a surprising result. If you try this in Java for instance, you would get `2` as the output. Note that even though both operands are integers, the result of the `/` operation is a floating point value, `2.5`. Python does what is expected by a programmer!

The puzzle after that tries to play with `5 + 5 * 6`. What would be the result of this expression? Will it be `5 + 5` or `10`, then `10 * 6`, which is `60`? Or, will it be `5` plus `5 * 6`, which is `5 + 30`, that's `35`?

```
1      >>> 5 + 5 * 6
2      35
```

The correct result is 35 .

Python decides this is based on the **precedence** of operators.

Operators in Python are divided into two sets as follows:

- `**` , `*` , `/` and `%` have higher precedence, or priority.
- `+` and `-` have a lower precedence.

Sub-expressions involving operators from { `*` , `/` , `%` , `**` } are evaluated before those involving operators from { `+` , `-` }

Let's try another small puzzle on precedence, with `5 - 2 * 2` . What would be the result of this? Will it be 6 , or 1 ? It's 1 , because `*` has a higher precedence than `-` . Thus `2 * 2` is 4 , and `5 - 4` gives us 1 .

```
1      >>> 5 - 2 * 2
2      1
```

Let's say we want to execute `5 - 2` , to give an output of 2 . How do we change the operator precedence?

You cannot really change the precedence, but you can add parentheses to group sub-expressions differently.

```
1      >>> (5 - 2) * 2
2      6
3      >>> 5 - ( 2 * 2 )
4      1
```

Parentheses have the highest precedence in Python, and can be used to override operator precedence. `(5 - 2)` gets calculated first, and the final result of the expression is `6`.

A positive thing about using parentheses is, that it makes expressions more readable. So even in situations such as `5 - 2 * 2`, where we know the result according to precedence, adding parentheses is good.

Summary

In this step, we went about solving a few puzzles about expressions, touching concepts such as:

- `SyntaxError` for incorrect operators
- White-space in expressions
- Floating Point division by default
- Operator Precedence
- Using parentheses

Step 06: Printing Text

In the previous step, we learned how to use expressions to compute values. In this step, let's see how we can actually print multiplication table entries, that are readable by the user.

Snippet-01: Printing Text

How do we go about printing a complete multiplication table entry? We want to print text such as `5 * 6 = 30`. But trying to do so, as we know it, gives us a `SyntaxError`. Clearly, there is a different way to print text, as compared to an expression.

```
1      >>> 5 * 6 = 30
2          File "<stdin>", line 1
3      SyntaxError: can't assign to operator
```

Let's first try to print a simple piece of text, `Hello`. Typing in this piece of code directly on Python Shell also gives us an error.

```
1      >>> Hello
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      NameError: name 'Hello' is not defined
```

Only expressions work that way, and `Hello` is not really an expression.

`"Hello"` is typically called a **string**, and represents the text of letters `'H'`, `'e'`, `'l'`, `'l'`, `'o'`. `"Hello"` is hence different from the number `5`.

There are a number of in-built functions in Python to help print strings. One of these is the `print()` function. Can you just say `print Hello`?

```
1      >>> print Hello
2      File "<stdin>", line 1
3          print Hello
4                  ^
5      SyntaxError: Missing parentheses in call to 'print'. Did you mean print(He
```

The Python compiler gives you an error, that says “missing parentheses”.

Will `print(Hello)` work?

```
1      >>> print (Hello)
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      NameError: name 'Hello' is not defined
```

Nope! Again, this one failed because you need to indicate that `"Hello"` is a string.

How do I indicate that `"Hello"` is a string? By putting it within double quotes.

Let's try `print ("Hello")`

```
1      >>> print ("Hello")
2      Hello
3      >>> print("Hello")
4      Hello
```

`print("Hello")` finally results in `"Hello"` being printed out. To be able to print `"Hello"`, the things we need to do are:

- Typing the method name `print`,
- open parentheses `(`,
- Followed by a double quote `"`,
- The text `Hello`,
- and another double quote `"`,
- finished off with a closed parentheses `)`.

What we have written here is called a **statement**, a simple piece of code to execute. As part of this statement, we are **calling a function**, named `print()`.

What exactly are we trying to print?

The text `"Hello"`, which is called a **parameter or argument**, to `print()`.

Now let's get back to what we wanted to do, which is to print `5 * 6 = 30`. The most basic version would be something of this kind, `print("5 * 6 = 30")`. Here, we are passing the entire value in the form of a string.

```
1      >>> print("5 * 6 = 30")
2      5 * 6 = 30
```

This prints the text on the console, as-is. The thing you need to understand here is, we aren't really calculating `30` using the formula `5 * 6`, but directly putting text `30` in here. That's called **hard-coding**.

In a later step, we will look at how to actually calculate the value and pass it in.

Summary

In this step, we:

- Understood that displaying text on the console is not the same as printing an expression value
- Learned about the `print()` function, that is used to print text in Python.
- Found a way to print the text `"5 * 6 = 30"` on the console, by hard-coding values in a string

Step 07: Puzzles On Utility Methods, And Strings

In the previous step, we learned how to print `5 * 6 = 30`. It was not a perfect solution, because we hard-coded everything. we used an in-built function named `print()`, passed a string to it, and invoked the method.

In this step, let's look at a number of puzzles related to in-built methods, their parameters, and strings in general.

For example, let's do `print("5 * 6")` , as in the previous step. What does this code result in?

```
1      >>> print("5*6")
2      5*6
3      >>> print('5*6')
4      5*6
```

It just prints the string `"5 * 6"` .

Let's say we try the code `print(5 * 6)` ,

```
1      >>> print(5*6)
2      30
```

Without the double quotes, `5 * 6` is an expression. What will be the output? `30` .

If you call `print()` with an expression argument, it prints the value of the expression. However, when we pass something within double quotes, it becomes a piece of text, printed as-

is.

An interesting thing to note is, that in Python you can use either double-quotes (") and ("), or single-quotes (') and (') with text values.

Let's look at a few other in-built methods within Python.

Consider `abs()` (which stands for absolute value), a method that accepts a numeric value. You can use `abs(10.5)`, passing `10.5` as a value to it, and it prints the absolute value of `10`.

```
1      >>> abs 10.5
2          File "<stdin>", line 1
3              abs 10.5
4                  ^
5      SyntaxError: invalid syntax
6      >>> abs(10.5)
7      10.5
```

If you pass in a string value, will it work? It complains, “`abs()` function will not work with a string, it only works with numeric values”.

```
1      >>> abs("10.5")
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      TypeError: bad operand type for abs(): 'str'
```

Let's say you want to use a function that computes “to the power of”, for instance “`2` to the power of `5`”. In Python, there's an in-built function named `pow()`, which does just what we need. To `pow()`, you can pass two parameters and calculate the result. How do you do that?

Will this work: `pow 2 5`? No, not at all. This code does not work as well: `pow(2 5)`. `pow(2, 5)` is the correct syntax.

```
1      >>> pow 2 5
2          File "<stdin>", line 1
```

```
3      pow 2 5
4          ^
5      SyntaxError: invalid syntax
6      >>> pow(2 5)
7          File "<stdin>", line 1
8              pow(2 5)
9                  ^
10     SyntaxError: invalid syntax
11     >>> pow(2, 5)
12     32
```

You'll see that `32` is printed.

Let's see another example, "`10` to the power of `3`". `pow(10,3)` is the alternative to saying `10 ** 3`. This gives us `1000`, similar to how `pow()` would.

```
1      >>> pow(10, 3)
2      1000
3      >>> 10 ** 3
4      1000
```

`max()` returns maximum in a set of numbers. `min()` function returns the minimum value.

```
1      >>> max(34, 45, 67)
2      67
3      >>> min(34, 45, 67)
4      34
```

These are some of the in-built functions in Python, and we saw how to call the in-built functions by passing in a varied number of parameters.

Python is case sensitive. So let's say I want of calculate `pow(2,5)`. So this would give me `32`. Now, what if I say capital `'P'` instead of small `'p'` here? `Pow(2,5)` would lead to an error.

```
1      >>> pow(2,5)
2      32
3      >>> Pow(2,5)
```

```
4      Traceback (most recent call last):
5          File "<stdin>", line 1, in <module>
6      NameError: name 'Pow' is not defined
```

The only things not case-sensitive in Python, are string values. Earlier we saw that the code `print("Hello")` displays the text "Hello". Inside a string, the text can be in any case. Hence, `print("hello")` displays "hello", with a small 'h'.

```
1      >>> print("Hello")
2      Hello
3      >>> print("hello")
4      hello
5      >>> print("hell0")
6      hell0
7      >>> print ("hell0" )
8      hell0
```

However inside your code, you need to be very particular about the case of function names, class names, variable names, and the like.

In your code, whitespace does not really matter. You can add space here and here, and you would still get the same output. However, in case of strings, whitespace does matter.

If we say `print("hello World")`, it would print "hello World", with a space in between. And if you do `print("hello World")` with three spaces, it would print the same. In expressions, white-space does not affect the output.

```
1      >>> print ( "hello World" )
2      hello World
3      >>> print ( "hello      World" )
4      hello      World
```

The last thing we want to look at, is an **escape sequence**. Let's say you want to print a double quote, " , in the code. If we were to do this: `print("Hello""")`, what would happen? The compiler says error!

```
1      >>> print("Hello\"")
2          File "<stdin>", line 1
3              print("Hello\"")
4                  ^
5      SyntaxError: EOL while scanning string literal
```

If you want to print a `"` inside a string, use an escape sequence. In Python, the symbol `'\'` is used as an **escape character**. On using `'\'` adjacent to the `"`, it prints `Hello"` (notice the trailing `"`). We have used the `'\'` to **escape** the `"`, by forming an *escape sequence* `\"`.

```
1  >>> print("Hello\"")
2  Hello"
3  >>>
```

The other reason why you would want to use a `'\'` is to print a `<NEWLINE>`. If you want to print `"Hello World"`, but with `"Hello"` on one line and `"World"` on the next, `'\n'` is the escape sequence to use.

```
1      >>> print("Hello\nWorld")
2      Hello
3      World
```

The other important escape sequence is `'\t'`, which prints a `<TAB>` in the output. When you do `print("Hello\tWorld")`, you can see the tab-space between `"Hello"` and `"World"`.

```
1      >>> print("Hello\tWorld")
2      Hello    World
```

Another useful escape sequence is `\\"`. If you want to print a `\`, then use the sequence `\\"`. You would see that it prints `Hello\World`. Think about what would happen if we put six `\`. Yes you're right! It would print this string: `"\\\\\\\"`.

```
1      >>> print("Hello\\World")
2      Hello\World
3      >>> print("Hello\\\\\\World")
4      Hello\\\\World
```

One of the things with Python is, it does not matter whether you use double quotes or single quotes to enclose strings. There are some interesting, and useful ways of using a combination of both, within the same string. Have a look at this call: `print("Hello'World")`, and notice the output we get. In a similar way, the following code will be accepted and run by the Python system: `print('Hello"World')`.

```
1      >>> print('Hello')
2      Hello"
3      >>> print("Hello'World")
4      Hello'World
5      >>> print("Hello\"World")
6      Hello"World
7      >>> print("Hello\"World")
8      Hello"World
```

The above two examples can be used as a tip by newbie programmers when they form string literals, and want to use them in their code:

- If the string literal contains one or more single quotes, then you can use double quotes to enclose it.
- However if the string contains one or more double quotes, then prefer to use single quotes to enclose it.

Summary

In this step, we:

- Explored a number of puzzles related to code involving:
 - Built-in functions for numeric calculations
 - The `print()` function to display expressions and strings
- Covered the following aspects of the above utilities:
 - Case-sensitive aspects of names and strings
 - The role played by whitespace

- The escape character, and common escape sequences

Step 08: Formatted Output With print()

In the previous step, we learned how to print a hard-coded string, such as "5 * 6 = 30".

In this step, let's try to replace the hard-coded 30 with a computed value.

Let's start with a simple scenario. Let's say we want to place that calculated value within a string, and display it. How do we do that?

Snippet-01: print() Formatted Output

format() method can be used to print formatted text.

Let's see an example:

```
1      >>> print("VALUE".format(5*2))
2      VALUE
```

We were expecting 10 to be printed, but it's actually printing VALUE .

How do we get 10 to be printed then?

```
1      >>> print("VALUE {0}".format(5*2))
2      VALUE 10
```

By having an open brace { , closed brace } , and by putting the index of the value between them. Here, the value is the first parameter, and its index will be 0 .

"VALUE {0}" is what we need.

Let's take another example. Suppose to the format() function, we pass three values: 10 , 20 and 30 .

Typically when we count positions or indexes, we start from `0`.

To print the first value, you need to pass in an index of `0`. To print the second value, pass an index of `1`.

```
1      >>> print("VALUE {0}".format(10,20,30))
2      VALUE 10
3      >>> print("VALUE {1}".format(10,20,30))
4      VALUE 20
5      >>> print("VALUE {2}".format(10,20,30))
6      VALUE 30
```

Now going back to our problem, we wanted to display `"5 * 6 = 30"`, but without hard-coding. Instead of `30`, we want the calculated value of `5 * 6`.

```
1      >>> print("5 * 6 = 30".format(5,6,5*6))
2      5 * 6 = 30
```

Let replace `"5 * 6 = 30"` with `"5 * 6 = {2}"`. `2` is the index of parameter value `5*6`.

```
1      >>> print("5 * 6 = {2}".format(5,6,5*6))
2      5 * 6 = 30
```

Cool! Progress made.

Let's replace `5 * 6` with the right indices - `{0} * {1}`.

```
1      >>> print("{0} * {1} = {2}".format(5,6,5*6))
2      5 * 6 = 30
```

The great thing about this, is now we can replace the values we passed to `print()` in the first place, without changing the indexes! So, we can display results for `5 * 7 = 35` and

`5 * 8 = 40`. We are now able to print `5 * 6 = 30`, `5 * 7 = 35`, `5 * 8 = 40`, and can do similar things for other table entries as well.

```
1      >>> print("{0} * {1} = {2}".format(5,7,5*7))
2      5 * 7 = 35
3      >>> print("{0} * {1} = {2}".format(5,8,5*8))
4      5 * 8 = 40
5      >>> print("{0} * {1} = {2}".format(5,8,5*8))
6      5 * 8 = 40
```

Summary

In this step, we:

- Discovered that Python provides a way to do formatted printing of string values
- Looked at the `format()` function, and saw how to call it within `print()`
- Observed how we could work only with the indexes of parameters to `format()`, and change the parameters we pass without changing the code

Step 09: Puzzles On `format()` and `print()`

In this step, let's look at a few puzzles related to the `format`, and the `print` methods.

Snippet-01: `format()` And `print()` Puzzles

Let's say we pass in additional values, such as: `5 * 8`, `5 * 9` and `5 * 10`. However, within the call to `format()`, we are only referring to the values at index `0`, index `1` and index `2`. The values at indexes `3` and `4` are not used at all. What would happen when we run the code?

```
1      >>> print("{0} * {1} = {2}".format(5,8,5*8,5*9,5*10))
2      5 * 8 = 40
```

Would this throw an error? No, it does not. You can see that the additional values which are passed in, are conveniently ignored.

Let's say instead of passing in a value of `2`, we pass `4`. What would happen?

```
1      >>> print("{0} * {1} = {4}".format(5,8,5*8,5*9,5*10))
2      5 * 8 = 50
```

`5 * 10` is the value at index `4`

Now let's take a different scenario. We remove all the parameters passed to `format()`. However, inside the call to `print()`, we continue to say `{0} * {1} = {4}`. So we are trying to print the value at index `4`, but are only passing two values to the function `format()`. What do you think will happen?

```
1      >>> print("{0} * {1} = {4}".format(5,8))
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      IndexError: tuple index out of range
```

It says `IndexError`, which means :"you are asking me to fetch the value at index `4`, but only passing in two values. How can I do what you want?"

Let's look at a few more things related to other data types. We try to format the following inside `print()`: `{0} * {1} = {2}`, and would pass in `2.5`, `2`, and `2.5 * 2`. Here, `2` is an integer value, but `2.5` is a floating point value. You can see that it prints `2.5 * 2 = 5.0`. So this approach of formatting values with `print()`, works also with floating point data as well.

```
1      >>> print("{0} * {1} = {2}".format(2.5,2,2.5*2))
2      2.5 * 2 = 5.0
```

Now, are there are other types of data that `format()` works with? Yes, strings can join the party.

Let's say over here, we do: `print("My name is {0}".format("Ranga"))`. What would happen?

```
1      >>> print("My name is {0}".format("Ranga"))
2      My name is Ranga
```

Index `0` will be replaced with the first parameter to `format()`.

Summary

In this step, we:

- Understood the behavior when the parameters passed to `format()`:
 - Exceed the indexes accessed by `print()`
 - Are less than the indexes accessed by `print()`
 - Are of type integer, floating-point or string

Step 10: Introducing Variables

We are slowly making progress toward our main goal, which is to print the `5` multiplication table.

In the first statement, we are printing `5 * 1 = 5`, and then changing the literals. To make it print `5 * 2 = 10`, we are changing `1` to `2`. Next, we are changing `2` to `3`. How do we make it a little simpler, so that our effort is reduced?

```
1      >>> print("{0} * {1} = {2}".format(5,1,5*1))
2      5 * 1 = 5
3      >>> print("{0} * {1} = {2}".format(5,2,5*2))
4      5 * 2 = 10
5      >>> print("{0} * {1} = {2}".format(5,3,5*3))
6      5 * 3 = 15
```

Let's try a different approach.

What would happen if you replace `1` with `index`, and `5 * 1` with `5 * index`, and try to run it?

It gives an error! It says: "index is not defined".

Let's try and fix this, and execute `index = 2`. What would happen?

```
>>> index = 2
```

Aha! This compiles.

```
1      >>> print("{0} * {1} = {2}".format(5,index,5*index))
2      5 * 2 = 10
```

And this statement is printing `5 * 2 = 10`.

Let's try something else. Let's make `index = 3`. What would happen?

```
1      >>> index = 3
2      >>> print("{0} * {1} = {2}".format(5,index,5*index))
3      5 * 3 = 15
```

The same statement on being run, prints `5 * 3 = 15`.

How can you check the value that `index` has? Just type in `index`.

```
1      >>> index
2      3
3      >>> print("{0} * {1} = {2}".format(5,index,5*index))
4      5 * 3 = 15
```

The `index` symbol we have used here, is what is called a **variable**.

In Python, it's also called a **name**.

You can see that the value `index` referring to, can change over the duration of a program.

Initially, `index` was referring to a value of `1`. later, `index` was referring to a value of `3`.

Now, think about how you would print the entire table. All that you need to do, is start from `1`, execute the same statement with `print()` and `format()`, to get output `5 * 1 = 5`. Next, Change the value of `index` to `2`, and then print the same statement. Next, `index = 3`, and print the same statement again.

```
1      >>> index = 1
2      >>> print("{0} * {1} = {2}".format(5,index,5*index))
3      5 * 1 = 5
4      >>> index = 2
5      >>> print("{0} * {1} = {2}".format(5,index,5*index))
6      5 * 2 = 10
7      >>> index = 3
8      >>> print("{0} * {1} = {2}".format(5,index,5*index))
9      5 * 3 = 15
```

With the same statement `print("{0} * {1} = {2}".format(5,index,5*index))`, we are able to print different values. The value of `index` varies, but the code remains the same!

Variables make the program much more easier to read, as well as more generic.

Snippet-02: Classroom Exercise On Variables

Let's do a simple exercise with variables.

We want to create three variables `a`, `b` and `c`. Let's initially give them some values, say a value of `5` to `a`, `6` to `b` and `7` to `c`.

We want to get output of this kind: `5 + 6 + 7 = 18`, without using the literal values.

You would want to use the values stored in the variables in `a`, `b` and `c`.

If you're hard-coding, the way to do it is with `print("5 + 6 + 7 = 18")`.

```
1      >>> a = 5
2      >>> b = 6
3      >>> c = 7
```

```
4      >>> print("5 + 6 + 7 = 18")
5      5 + 6 + 7 = 18
6      >>> print("5 + 6 + 7 = 18".format(a,b,c,a+b+c))
7      5 + 6 + 7 = 18
```

The way you can do that is with code like this:

```
print("{0} + {1} + {2} = {3}".format(a,b,c,a+b+c)) .
```

```
1      >>> print("{0} + {1} + {2} = {3}".format(a,b,c,a+b+c))
2      5 + 6 + 7 = 18
```

How do you confirm we are accessing values stored in the variables?

Let's change the values of `a`, `b` and `c`. Let's make `a = 6`, `b = 7`, and `c = 8`. Execute same statement.

```
1      >>> a = 6
2      >>> b = 7
3      >>> c = 8
4      >>> print("{0} + {1} + {2} = {3}".format(a,b,c,a+b+c))
5      6 + 7 + 8 = 21
```

You can see the magic of variables at play here! Based on what values these variables are referring to, you can see that the output of the print statement changes.

Summary

In this step, we:

- Were introduced to variables, or names, in Python
- Observed how we could pass in values of variables to the `format()` function

Step 11: Puzzles On Variables

In the previous step, we were introduced to the concept of variables in Python.

We will start with looking at a few puzzles.

Snippet-01: Puzzles On Variables

What if I try to refer to a variable which is not yet created?

```
1      >>> count
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      NameError: name 'count' is not defined
5      >>> print(count)
6      Traceback (most recent call last):
7          File "<stdin>", line 1, in <module>
8      NameError: name 'count' is not defined
```

Before using a variable, you need to have it assigned a value. If you have not defined a variable before, then you cannot use it. Consider `print(count)`, it does not know what `count` is. So it would throw an error, saying: “`count` is not defined, I have no idea what `count` is.”

Once you assign a value to a variable, you can use it.

```
1      >>> count = 4
2      >>> print(count)
3      4
```

The statement `count = 4` where we are creating a variable named `count` for the first time, is called a **variable definition**.

This is the first time you’re referring to a variable, and assigning a value to it.

Python will create a variable in its memory.

Variable names are case sensitive. `count` and `Count` are not the same thing.

```
1      >>> Count
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
```

```
4      NameError: name 'Count' is not defined
5      >>> count
6      4
```

There are rules to follow while naming variables.

All variable names should either start with an alphabet , or an underscore `_ . count , _count` are valid. `1count` is invalid.

```
1      >>> 1count = 5
2          File "<stdin>", line 1
3              1count = 5
4                  ^
5          SyntaxError: invalid syntax
6      >>> count = 5
7      >>> _count = 5
8      >>> 1count
9          File "<stdin>", line 1
10         1count
11             ^
12         SyntaxError: invalid syntax
13     >>> 2count
14         File "<stdin>", line 1
15         2count
16             ^
17     SyntaxError: invalid syntax
```

After the first symbol, you can also use a numeral in variable names.

```
>>> c12345 = 5
```

To summarize the rules for naming variables.

- This should start with an alphabet (a capital or a small alphabet) or underscore.
- Starting the second character, it can be alphabet, or underscore, or a numeric value.

Summary

In this step, we:

- Understood that a variable needs to be defined before it is used
- Learned that there are certain rules to be followed while giving names to variables

Step 12: Introducing Assignment

In this step, we will look at an important concept in Python, called **assignment**. In previous steps, we created variables, like `i = 5`.

Snippet-01: Introducing Assignment

You can create other variables using whatever value `i` is referring to. If we say `j = i`, what would happen?

```
1      >>> i = 5
2      >>> j = i
3      >>> j
4      5
```

`j` would start referring to the same value that `i` is referring to. This statement is called an **assignment**.

Let's try `j = 2 * i`.

```
1      >>> j = 2 * i
2      >>> j
3      10
```

`j` refers to a value of `10`

`=` has a different meaning in programming compared to mathematics.

In mathematics, When we execute `j = i`, it means `j` and `i` are equal.

In programming, the value of the expression on right hand side is assigned to the variable on the right hand side. Can you use a constant on the left hand side of an assignment? The answer is “No”!

```
1      >>> 5 = j
2          File "<stdin>", line 1
3      SyntaxError: can't assign to literal
```

The Python Shell throws an error, saying “Can’t assign to literal”, as `5` is a literal.

Let’s create a couple of variables. `num1 = 5` and `num2 = 3`. We would want to add these and create a fresh variable. Let’s say the name of the variable is `sum`.

```
1      >>> num1 = 5
2      >>> num2 = 3
3      >>> sum = num1 + num2
4      >>> sum
5      8
```

Create 3 variables `a`, `b` and `c` with different values and calculate their sum.

```
1      >>> a = 5
2      >>> b = 6
3      >>> c = 7
4      >>> sum = a + b + c
5      >>> sum
6      18
```

We have just seen the mechanics of how assignment works in Python.

Summary

In this step, we:

- Learned what happens when you assign a value to a variable, which may or may not exist
- Discovered that literal constants cannot be placed on the left hand side of the assignment(`=`) operator

Step 13: Introducing Formatted Printing

Until now, we have been using the `format()` method to format and print values. Let's see a better approach to printing values.

This is the approach we used until now.

```
1      >>> a = 1
2      >>> b = 2
3      >>> c = 3
4      >>> sum = a + b + c
5      >>> print("{0} + {1} + {2} = {3}".format(a, b, c ,sum))
6      1 + 2 + 3 = 6
```

Python has the concept of formatted strings. The syntax to use a formatted string is very simple - `f""`.

If we want to print the value of a variable `a`, we can use `{a}` in the text.

```
1      >>> print(f"")
2      >>> print(f"value of a is {a}")
3      value of a is 1
4      >>> print(f"value of b is {b}")
5      value of b is 2
```

The variable within braces is replaced by its value.

You can use expressions in a formatted string. Example below uses `{a+b}`.

```
1      >>> print(f"sum of a and b is {a + b}")
2      sum of a and b is 3
```

This feature was introduced in a Python 3 release.

Let's get back to the original problem we wanted to solve: printing `5 + 6 + 7 = 18`, using formatted strings.

```
1      >>> print(f"{a} + {b} + {c} = {sum}")
2      1 + 2 + 3 = 6
```

You can see how easy it turns out to be!

Step 14: The PMT-Challenge Revisited

We want to print the `5`-table from `5 * 1 = 5` onward, until we reach to `5 * 10 = 50`. The best solution we have right now, is shown below:

Snippet-01:

```
1      >>> index = 1
2      >>> print("{0} * {1} = {2}".format(5,index,5*index))
3      5 * 1 = 5
4      >>> index = 2
5      >>> print("{0} * {1} = {2}".format(5,index,5*index))
6      5 * 2 = 10
7      >>> index = 3
8      >>> print("{0} * {1} = {2}".format(5,index,5*index))
9      5 * 3 = 15
10     >>> index = 4
11     >>> print("{0} * {1} = {2}".format(5,index,5*index))
12     5 * 4 = 20
```

Can we do something, to make sure that the code remains the same all the time, but the `index` value gets updated?

```
1      >>> index = index + 1
2      >>> print("{0} * {1} = {2}".format(5,index,5*index))
3      5 * 5 = 25
4      >>> index = index + 1
5      >>> print("{0} * {1} = {2}".format(5,index,5*index))
6      5 * 6 = 30
7      >>> index = index + 1
8      >>> print("{0} * {1} = {2}".format(5,index,5*index))
9      5 * 7 = 35
```

We used `index = index + 1` to increment `index` value.

If we execute these same two statements again and again, we can print the entire table! This is exactly what loops help us do: execute the same statements repeatedly.

The simplest loop available in Python is the **for loop**.

When we run a `for` loop, we need to specify the range of values - `1` to `10` or `1` to `20`, and so on. `range()` function helps us to specify a range of values.

```
1      >>> range(1,10)
2      range(1, 10)
```

The syntax of the `for` loop is: `for i in range(1, 10): ...`. Here, `i` is the name of the **control variable**. In Python, you need to put a colon, '`:`', and in the next line give indentation.

```
1      >>> for i in range(1,10):
2          ...     print(i)
3          ...
4          1
5          2
6          3
7          4
8          5
9          6
10         7
11         8
12         9
```

You would see that it prints from `1` to `9`.

When we run a loop in `range(1, 10)`, `1` is **inclusive** and `10` is **exclusive**. The loop runs from `1` to the value before `10`, which is `9`.

The leading whitespace before `print(i)` is called **indentation**. We'll talk about indentation later, when we talk about puzzles related to the `for` loop.

How can you extend this concept to solving our *PMT-Challenge* problem?

```
1      >>> print(f"5 * {index} = {5*index}")
2      5 * 7 = 35
```

What we were doing earlier, was calling `print()` with a formatted string. Now we want to print this statement for different values of `i`.

How can you do that?

Let's start with a simple example.

```
1      >>> for i in range(1,11):
2          ...     print(f"{i}")
3          ...
4          1
5          2
6          3
7          4
8          5
9          6
10         7
11         8
12         9
13         10
```

`print(f"{i}")` prints the value of `i`.

Now, how do we get it to print `5 * 1 = 5` to `5 * 10 = 50` ?

```
1      >>> for i in range(1,11):
2          ...     print(f"5 * {i} = {5 * i}")
3          ...
4          5 * 1 = 5
5          5 * 2 = 10
6          5 * 3 = 15
7          5 * 4 = 20
8          5 * 5 = 25
9          5 * 6 = 30
```

```
10      5 * 7 = 35
11      5 * 8 = 40
12      5 * 9 = 45
13      5 * 10 = 50
14      >>> 5 * 4 * 50
15      1000
```

`print(f"5 * {i} = {5 * i}")` prints a specific multiple of 5.

Step 15: Loops

In a previous step, we took a major step in programming. We wrote our first for loop with Python. In this step, let's try a few puzzles to understand the for loop even further.

The syntax of the for loop we looked at earlier was:

```
1  for i in range(1, 10):
2      print(i)
```

Snippet-01:

Let's say we write a `for` loop, but don't give a `:` after the `range()` method, to close the first line. What would happen?

```
1      >>> for i in range(1,10)
2          File "<stdin>", line 1
3              for i in range(1,10)
4                  ^
5      SyntaxError: invalid syntax
```

Invalid syntax. A `:` is mandatory within the `for` loop syntax.

Let's provide a `:` and in the next line, use `print(i)` without space before it (without indentation).

```
1      >>> for i in range(1,10):
```

```
2     ... print(i)
3         File "<stdin>", line 2
4             print(i)
5                 ^
6 IndentationError: expected an indented block
```

Most other programming languages use open brace `{` and closed brace `}` as delimiters in a `for` loop. However, Python uses indentation to identify which code is part of a `for` loop, and which is not. So if we are writing the body of a `for` loop, we must use indentation, and leave atleast a single `<SPACE>`.

```
1      >>> for i in range(1,10):
2          ...     print(i)
3          ...
4          1
5          2
6          3
7          4
8          5
9          6
10         7
11         8
12         9
```

How do we execute two lines of code as part of the `for` loop?

```
1      >>> for i in range(1,10):
2          ...     print(i)
3          ...     print(2*i)
4          ...
5          1
6          2
7          2
8          4
9          3
10         6
11         4
12         8
13         5
14         10
15         6
16         12
17         7
```

```
18      14
19      8
20      16
21      9
22      18
```

We are indenting both statements with a space - `print(i)` and `print(2*i)`.

When for loop has only one line of code, you can specify it right after the `:`:

```
1      >>> for i in range(2,5): print(i)
2      ...
3      2
4      3
5      4
```

However, this is not considered to be a good programming practice. Even though you may want to execute just one statement in a `for` loop, indentation on a new line is recommended.

Another best practice is to use four `<SPACE>`s for indentation, instead of just two. This would give clear indentation of the code.

```
1      >>> for i in range(2,5):
2          ...     print(i)
3          ...
4          2
5          3
6          4
```

Anybody who looks at the code immediately understands that this `print()` is part of the `for` loop.

Let's say you only want to print the odd numbers till `10`, which are `1`, `3`, `5`, `7` and `9`. The `range()` function offers an interesting option.

```
1      >>> for i in range (1,11,2):
```

```
2     ...     print(i)
3     ...
4     1
5     3
6     5
7     7
8     9
```

In `for i in range(1, 11, 2)`, we pass in a third argument, called a *step*. After each iteration, the value of `i` is increment by `step`.

Summary

In this step, we:

- Looked at a few puzzles about the `for` loop, which lay emphasis on the following aspects of `for`:
 - The importance of syntax elements such as the colon
 - Indentation
 - Variations of the `range()` function

Step 16: Programming Exercise PE-BA-02

In the previous step, after initially exploring the Python `for` loop, we looked at a number of puzzles.

In this step, let's look at a few exercises.

Exercises

1. Print the even numbers up to 10. We would want to print 2 4 6 8 10, using a `for` loop.
2. Print the first 10 numbers in reverse
3. Print the first 10 even numbers in reverse
4. Print the squares of the first 10 numbers
5. Print the squares of the first 10 numbers, in reverse
6. Print the squares of the even numbers

Solution 1

Instead of starting with `1`, we need to start with `2`. Each time, `i` it would be incremented by `2`, and `2 4 6 8` and `10` would be printed.

```
1      >>> for i in range (2,11,2):
2          ...     print(i)
3          ...
4          2
5          4
6          6
7          8
8          10
```

Solution 2

We would want to print the numbers in reverse. Think about how you would do that using the `range()` function. We'd want go from `10`, `9`, `8`, and so on up to `1`.

```
1      >>> for i in range (10,0,-1):
2          ...     print(i)
3          ...
4          10
5          9
6          8
7          7
8          6
9          5
10         4
11         3
12         2
13         1
```

The value to start with is `10`. As we discussed earlier, the end value is exclusive. So to print from `10` to `1`, we want to end one value which is `0`. `range(10, 0)` seems to be what we need.

Usually these step value is positive, but we need to go backwards from `10`. Hence, we would give a step value of `-1`.

Solution 3

Now, let's print the first 10 even numbers in reverse.

```
1      >>> for i in range (20,0,-2):
2          ...
3          ...
4          20
5          18
6          16
7          14
8          12
9          10
10         8
11         6
12         4
13         2
```

Solution 4

Next, we would want to print the squares of the first 10 numbers.

```
1      >>> for i in range (1,11):
2          ...
3          ...
4          1
5          4
6          9
7          16
8          25
9          36
10         49
11         64
12         81
13         100
```

Solution 5

Let's print the squares in the reverse order.

```
1      >>> for i in range (10,0,-1):
2          ...
3          ...
```

```
4      100
5      81
6      64
7      49
8      36
9      25
10     16
11     9
12     4
13     1
```

Solution 6

Print the squares of the even numbers. How to do that?

```
1      >>> for i in range (10,0,-2):
2          ...
3          ...
4          100
5          64
6          36
7          16
8          4
```

The key part is using a step of -2

We leave it as an exercise for you, to print squares of odd numbers.

Summary

In this video, we: * Tried out a few exercises involving the for loop, by playing around with printing sequences of numbers.

- Used the for loop to simplify the solution to the *PMT-Challenge* problem.

Step 17: Review: The Basics Of Python

It must have been a roller-coaster ride to solve the multiplication table challenge so far. If you're new to programming, there are a wide range of topics and concepts, that you would have learned during this small journey.

Let's quickly revise the important concepts we have learned during this small journey.

- `1`, `11`, `5`, ... are all called literals because these are constant values. Their values don't really change. Consider `5 - 4 - 50`. This is an expression. `-` is an operator, and `5`, `4` and `50` are operands.
- The name `i` in `i = 1`, is called a variable. It can refer to different values, at different points in time.
- `range()` and `print()` are in-built Python functions.
- Every complete line of code is called statement. The specific statement `print()`, is invoking a method. The other statement which we looked at earlier, was an assignment statement. `index = index + 1` would evaluate `index + 1`, and have the `index` variable refer to that value.
- The syntax of the `for` loop was very simple. `for var in range(1, 10) : ...`, followed by statements you would want to execute in a loop, with indentation. For the sake of indentation we left four `<SPACE>`s in front of each statement inside the `for` loop.

So that, in a nutshell, is what we have learned over the course of our first section.

Chapter 03 - Introducing Methods

In the last section, we introduced you to the basics of python. We learned those concepts by applying them to solve the *PMT-Challenge* problem. The code below is what we ended up with as we solved that challenge.

Snippet-01: Current Solution To PMT-Challenge

```
1      >>> for i in range (1,11):
2          ...     print(f"8 * {i} = {8 * i}")
```

If we wanted to change the code to print the `7` table, we need to change the value `7` used in the for loop, to `8`. It's simple, but still not as friendly as you would like.

```
1      >>> for i in range (1,11):
2          ...     print(f"7 * {i} = {7 * i}")
```

To print a `7` table, it would be awesome if we could say `print_multiplication_table`, and give a value of 7 beside it, and it would do the rest:

```
1      >>> print_multiplication_table(7)
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      NameError: name 'print_multiplication_table' is not defined
5      >>> print_multiplication_table(8)
6      Traceback (most recent call last):
7          File "<stdin>", line 1, in <module>
8      NameError: name 'print_multiplication_table' is not defined
```

Similarly, `print_multiplication_table(8)`, could print the multiplication table for `8` !

To be able to do this, we need to create a **method**, or a **function**. Creating a method makes the code *reusable*, and we can invoke that method very easily by passing *arguments*.

In this section, we take an in-depth look at methods.

Step 01: Defining Your First Method

Methods are very important building blocks in Python programming. In this step, we will create a simple method that prints `"Hello World"`, twice.

Snippet-01:

When we talk about a method, we need to give it a name. We are already using an in-built Python method here, which is `print()`.

```
1      >>> print("Hello World")
2      Hello World
3      >>> print("Hello World")
4      Hello World
```

Similar to that, we need to give a name to our body of code. Let's say the name is `print_hello_world_twice`.

The syntax to create a method in Python is straightforward:

- At the start, use the keyword `def` followed by a space.
- Followed by name of the method - `print_hello_world_twice` .
- Add a pair of parenthesis: `()` .
- This is followed by a colon `:` (similar to what we used in a `for` loop).

```
1  >>> def print_hello_world_twice():
2      ...     print("Hello World")
3      ...     print("Hello World")
4      ...
```

All statements in a method should be indented. The two `print("Hello World")` are indented. So, they are part of the method body.

`print_hello_world_twice()` defines a method, and it has certain code inside its body.

How do we call this method? Is it sufficient to say `print_hello_world_twice` ?

```
1      >>> print_hello_world_twice
2      <function print_hello_world_twice at 0x10a71ef28>
```

Python Shell says, there's a function defined with that specific name.

How do we execute a method? Very simple! Add a pair of parentheses to the name, `()` !

```
1      >>> print_hello_world_twice()
2      Hello World
3      Hello World
4      >>> print_hello_world_twice()
5      Hello World
6      Hello World
```

Now, we are able to run the method.

Summary

In this step, we:

- Learned we can define our own methods in the code we write
- Understood how to define a method, and all its syntax elements
- Saw how we can invoke a method we write

Step 02: Programming Exercise PE-MD-01

We will now leave you with two exercises, based on what we have learned about methods so far.

Exercises

1. Write a method called `print_hello_world_thrice()`. It should print "Hello World" thrice to the output. Define this method, and also invoke it.
2. Write and execute a method, that prints four statements:
 1. "I have created my first variable."
 2. "I've created in my first loop."
 3. "I've created my first method."
 4. "I am excited to learn Python." You need to print these four statements on four consecutive lines.

Solutions

Solution 1

```
1      >>> def print_hello_world_thrice():
2          ...     print("Hello World")
3          ...     print("Hello World")
4          ...     print("Hello World")
5          ...
6      >>> print_hello_world_thrice()
7      Hello World
8      Hello World
9      Hello World
```

Solution 2

```
1      >>> def print_your_progress():
2          ...     print("Statement 1")
3          ...     print("Statement 2")
4          ...     print("Statement 3")
5          ...     print("Statement 4")
6          ...
7      >>> print_your_progress()
8      Statement 1
9      Statement 2
10     Statement 3
11     Statement 4
12
13     def print_your_progress():
14         print("Statement 1")
15         print("Statement 2")
16         print("Statement 3")
17         print("Statement 4")
```

For convenience, we have changed the exact text we need to print. Call this method with the syntax `print_your_progress()`, and you're able to execute its code.

Now try another exercise. We want to print "Statement 1", "Statement 2", "Statement 3" and "Statement 4" on different lines, using just one print statement. How can you do that?

```
1      >>> def print_your_progress():
2          ...     print("Statement 1\nStatement 2\nStatement 3\nStatement 4")
3          ...
4      >>> print_your_progress()
5      Statement 1
6      Statement 2
7      Statement 3
8      Statement 4
```

We are using the newline character `\n`.

Let's look at the difference between defining and executing a method.

When we are writing a method definition, we are writing the code as part of its body. It has a specific syntax, and starts with the `def` keyword.

A definition by itself cannot cause the code in its body to be executed.

```
print_your_progress()
```

 represents a method call. The code inside the method is executed.

Summary

In this step, we:

- Implemented solutions to a few exercises that test our understanding of Python methods.

We touched concepts such as:

- Defining a method body
- The way to invoke a method, to run its code
- The difference between the two

Step 03: Passing Parameters To Methods

In the previous step, we created methods. We defined `print_hello_world_twice()`, and this printed "Hello World" twice. In this step, let's talk about *method arguments*, or *parameters*.

Snippet-01:

```
1      >>> print_hello_world_twice()
2      Hello World
3      Hello World
4      >>> print_hello_world_thrice()
5      Hello World
6      Hello World
7      Hello World
```

Earlier, we wrote code for `print_hello_world_thrice()`, which prints the message three times.

Let's say you want to print it five times. You would need to write another method that does what you need. Doesn't that seem monotonous?

Instead of that, Won't it be great if I can call the method by the same name, say `print_hello_world(5)`, and it would print "Hello World" five times?

The `5` which we are passing here is called an **argument**.

How do we define our method to accept this argument?

Let's call our argument `no_of_times`. If you have any experience with other programming languages, they generally need you to specify the parameter type. Something like `This parameter is an integer/float/string, or other types`. But Python does not require parameter type.

```
1      >>> def print_hello_world(no_of_times):  
2          ...     print("Hello World")  
3          ...     print(no_of_times)  
4          ...
```

Although we are not doing exactly what we set out to, let's see what would happen. What would happen if we say `print_hello_world()` ?

```
1      >>> print_hello_world()  
2      Traceback (most recent call last):  
3          File "<stdin>", line 1, in <module>  
4      TypeError: print_hello_world() missing 1 required positional argument: 'no
```

Error! Something like "Hey, you have created `print_hello_world` with a parameter, but not passing anything in here! Go ahead and pass a value". Let's pass in a value, such as `5`.

```
1      >>> print_hello_world(5)  
2      Hello World  
3      5  
4      >>> print_hello_world(10)  
5      Hello World  
6      10  
7      >>> print_hello_world(100)  
8      Hello World  
9      100
```

With `print_hello_world(5)`, you can see `"Hello World"` and `5` being printed. We are now able to define this method to accept a value, and print that value by invoking it. You can pass in any value, such as `10`, `100`, or others.

Now think of a different solution for this method, where you don't repeat the same piece of code to print "Hello World". Consider `print_hello_world(5)`, it should still print "Hello World" 5 times. How do you do that?

Think about using something along the lines of a loop.

Snippet-02:

For now, what we are doing is we are printing "Hello World" 10 times.

```
1      >>> def print_hello_world(no_of_times):  
2          ...      for i in range(1,10):  
3              ...          print("Hello World")  
4              ...  
5  
6      >>> print_hello_world(5)  
7      Hello World  
8      Hello World  
9      Hello World  
10     Hello World  
11     Hello World  
12     Hello World  
13     Hello World  
14     Hello World  
15     Hello World
```

Our method call `print_hello_world(5)` now prints "Hello World" 10 times.

However just print the message 5 times. We need to make use of the parameter `no_of_times` inside the `for` loop as well.

```
1      >>> def print_hello_world(no_of_times):  
2          ...      for i in range(1,no_of_times):  
3              ...          print("Hello World")  
4              ...  
5  
6      >>> print_hello_world(5)  
7      Hello World  
8      Hello World  
9      Hello World  
10     Hello World
```

Now let's execute the method again. You can see that it's printing 4 times only.

Why is it not printing 5 times?

That's because no_of_times as a second parameter to range() is exclusive.

```
1      >>> def print_hello_world(no_of_times):  
2          ...     for i in range(1,no_of_times+1):  
3              ...         print("Hello World")  
4          ...  
5      >>> print_hello_world(5)  
6      Hello World  
7      Hello World  
8      Hello World  
9      Hello World  
10     Hello World
```

Great, it's now printing the message 5 times!

```
1      >>> print_hello_world(7)  
2      Hello World  
3      Hello World  
4      Hello World  
5      Hello World  
6      Hello World  
7      Hello World  
8      Hello World
```

If you pass a different argument like 7, the message is displayed 7 times.

Something you need to always be cautious about in Python, is the indentation. Over here, the for loop is part of the method body. So we have extra indentation for it. The print is part of the for loop body. So guess what, even more indentation for that code.

Summary

In this step, we:

- Learned how to pass arguments to a method

- Understood that the method definition needs to have parameters coded in
- Observed that arguments passed during a method call can be accessed inside a methods body

Step 04: Classroom Exercise CE-MD-01

In this step, Let's look at a few exercises related to the method parameter.

Exercises

1. Write a method called `print_numbers()` , that would print all successive integers from `1` to `n` .
2. The second one is to write a method called `print_squares_of_numbers()` , that prints squares of all successive integers from `1` to `n` .

Solutions

Solution 1

```
1      >>> def print_numbers(n):
2          ...     for i in range(1, n+1):
3              ...         print(i)
4          ...
5      >>> print_numbers(5)
6          1
7          2
8          3
9          4
10         5
11         >>>
```

If you are programming in other languages such as Java, you are used to naming methods in this way: `printNumbers()` . This convention is popularly known as “Camel Case”.

That's NOT how Python programmers name their methods. Pythonic way is to use underscore `_` to separate words in the method name, as in `print_numbers()` .

Solution 2

Let's define `print_squares_of_numbers()` . This would be very similar to `print_numbers()` , working with the same range. Only, we need to say `print(i*i)` .

```
1      >>> def print_squares_of_numbers(n):
2          ...     for i in range(1, n+1):
3              ...         print(i*i)
4          ...
5      >>> print_squares_of_numbers(5)
6      1
7      4
8      9
9      16
10     25
```

How is a parameter different from an argument?

- Inside the definition of the method, the name within parentheses is referred to as a **parameter**. In our recent exercise, `n` is a parameter, because it's used in the definition of `print_squares_of_numbers` .
- When you are passing a value to a method during a method call, say `5` , that value is called an **argument**.
- Don't worry too much about it. Just follow this convention for now:
 - In the method call, call it an *argument*.
 - In a method definition, call it a *parameter*.

Summary

In this step, we looked at a few simple exercises related to passing method arguments

Step 05: Methods With Multiple Parameters

In this step, let's look at creating a method with multiple parameters.

Snippet-01:

`print_hello_world` accepts one parameter and prints "Hello World" the specified number of times.

```
1      >>> def print_hello_world(no_of_times):
2          ...     for i in range(1,no_of_times+1):
3              ...         print("Hello World")
4          ...
```

Let's say we want to print another piece of text `Welcome To Python`, a specified number of times. How do you do that?

You can always create another method similar to the first one, such as

```
print_welcome_to_python(no_of_times)
```

 and print the necessary text inside.

However, is that what a good programmer does?

A good programmer tries to create a more generic solution.

```
1      >>> def print_string(str, no_of_times):
2          ...     for i in range(1,no_of_times+1):
3              ...         print(str)
4          ...
5      >>> print_string("Hello World", 3)
6      Hello World
7      Hello World
8      Hello World
```

The good programmer that you are, you created a new method called

```
print_string(str, no_of_times)
```

 accepting a text parameter, in addition to `no_of_times`.

Syntax rules for method parameters are quite strict. If we say

```
print_string("Welcome to Python")
```

 and run it, we get an error! Python Shell says: "I need `no_of_times` to be present in here".

```
1      >>> print_string("Welcome to Python")
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4      TypeError: print_string() missing 1 required positional argument: 'no_of_
```

Let's say you want to assign default values for `str` and `no_of_times` in `print_string()`. By default, we want to always print `"Hello World"`, and that too 5 times.

The Python language makes this very easy.

`def print_string(str = "Hello World", no_of_times=5)`. The rest of the method remains the same.

```
1      >>> def print_string(str="Hello World", no_of_times=5):
2          ...     for i in range(1,no_of_times+1):
3              ...         print(str)
4          ...
```

Now you can call `print_string()`, and `"Hello World"` is displayed 5 times.

```
1      >>> print_string()
2      Hello World
3      Hello World
4      Hello World
5      Hello World
6      Hello World
```

If it's `print_string("Welcome To Python")`, what does it do? It prints `"Welcome To Python"`, 5 times.

```
1      >>> print_string("Welcome to Python")
2      Welcome to Python
3      Welcome to Python
4      Welcome to Python
5      Welcome to Python
6      Welcome to Python
```

Consider `print_string("Welcome to Python", 8)`, it would print that string 8 times.

```
1      >>> print_string("Welcome to Python", 8)
2      Welcome to Python
```

```
3     Welcome to Python
4     Welcome to Python
5     Welcome to Python
6     Welcome to Python
7     Welcome to Python
8     Welcome to Python
9     Welcome to Python
```

Isn't that cool!

Summary

In this step, we:

- Looked at how to pass multiple parameters to a method, starting with two arguments
- Learned how you can define default values for those parameters
- Observed we could pass default arguments for none, some or all of those parameters

Step 06: Back To Multiplication Table - Using Methods

Let's get back to our original goal, of why we needed methods. We wanted to create a multiplication table for a number, and observed that each time we needed to we needed change that number, we were forced to make a change in the code. This is not something we liked, and that's why we started investigating how methods can be used.

In this step, Let's try our hand at creating a multiplication table method.

Snippet-01:

```
1      >>> for i in range (1,11):
2          ...     print(f"7 * {i} = {7 * i}")
```

Let's define a method called `print_multiplication_table()` , and pass in a parameter to it.

```
1      >>> def print_multiplication_table(table):
2          ...     for i in range(1,11):
3              ...         print(f"{table} * {i} = {table * i}")
4          ...
```

```
5      >>> print_multiplication_table(7)
6      7 * 1 = 7
7      7 * 2 = 14
8      7 * 3 = 21
9      7 * 4 = 28
10     7 * 5 = 35
11     7 * 6 = 42
12     7 * 7 = 49
13     7 * 8 = 56
14     7 * 9 = 63
15     7 * 10 = 70
```

Now you have the entire multiplication table for `7`.

You can then call `print_multiplication_table()` with arguments `8`, `9`, and so on, by simply changing the `table` argument value.

We now want to create even better `print_multiplication_table()` method.

We want to control the start point, as well as the end point, in the call to `range()`. We want to say `print_multiplication_table(7, 1, 6)`, to print the `7` table with entries from `1` to `6`. How can you do that?

```
1      >>> def print_multiplication_table(table, start, end):
2          ...      for i in range(start, end+1):
3              ...          print(f"{table} * {i} = {table * i}")
4          ...
5      >>> print_multiplication_table(7, 1 , 6)
6      7 * 1 = 7
7      7 * 2 = 14
8      7 * 3 = 21
9      7 * 4 = 28
10     7 * 5 = 35
11     7 * 6 = 42
```

Simple! Define those range limits as additional parameters!

The other thing we can obviously do, is have default values for the `start`, and the `end`.

```
1      >>> def print_multiplication_table(table, start=1, end=10):
```

```
2     ...     for i in range(start, end+1):
3     ...         print(f"{table} * {i} = {table * i}")
4 ...
5
6     >>> print_multiplication_table(7)
7     7 * 1 = 7
8     7 * 2 = 14
9     7 * 3 = 21
10    7 * 4 = 28
11    7 * 5 = 35
12    7 * 6 = 42
13    7 * 7 = 49
14    7 * 8 = 56
15    7 * 9 = 63
16    7 * 10 = 70
```

Calling `print_multiplication_table(7)` would give us entries from `7 * 1 = 7` to `7 * 10 = 70`.

Now you can actually send out this method, to your friends, who would find it easy to use, and cool!

Summary

In this step, we:

- Learned how to define a method to print the multiplication table for a number
- Looked at how to enhance this method to make table printing more flexible
- Further enhanced that method to accept default arguments while printing a table

Step 07: Indentation Is King

In Python, indentation denote blocks of code. So if you want to put something in a `for` loop, or outside it, proper indentation would be sufficient. In this step, let's explore indentation in depth. Let's start by creating a simple method.

Snippet-01:

```
1     >>> def method_to_understand_indentation():
2     ...     for i in range(1,11) :
3     ...         print(i)
4     ...
```

```
5      >>> method_to_understand_indentation()
6      1
7      2
8      3
9      4
10     5
11     6
12     7
13     8
14     9
15     10
```

Consider the code below: `print(5)` is indented at the same level as `for loop`.

```
1      >>> def method_to_understand_indentation():
2          ...      for i in range(1,11) :
3              ...          print(i)
4              ...          print(5)
5          ...
```

You can see that `print(5)` is called only once. It is not part of the `for loop`.

```
1      >>> method_to_understand_indentation()
2      1
3      2
4      3
5      4
6      5
7      6
8      7
9      8
10     9
11     10
12     5
```

Let's change the code in this method a bit. `print(5)` is indented the same way as `print(i)`

```
1      >>> def method_to_understand_indentation():
2          ...      for i in range(1,11) :
3              ...          print(i)
```

```
4      ...
5      ...
```

`print(5)` is part of the for loop. It is executed 10 times.

```
1      >>> method_to_understand_indentation()
2      1
3      5
4      2
5      5
6      3
7      5
8      4
9      5
10     5
11     5
12     6
13     5
14     7
15     5
16     8
17     5
18     9
19     5
20     10
21     5
```

Whether we're talking about loops, methods or conditionals, proper indentation is very important in Python.

We indicate a block of code, by having all lines of that block at the same indentation level. There are no specific delimiters like for instance a pair of braces `{...}`, as in other programming languages.

Summary

In this step, we:

- Ran through a few examples to see how indentation works in Python

Step 08: Puzzles on Methods - Named Parameters

In this step, let's look at a variety of puzzles related to methods.

Snippet-01:

Consider the following method: I would want to print the default string 6 times. How do we do it?

```
1      >>> def print_string(str="Hello World", no_of_times=5):
2          ...      for i in range(1,no_of_times+1):
3              ...          print(str)
4          ...
5      >>> print_string()
6      Hello World
7      Hello World
8      Hello World
9      Hello World
10     Hello World
```

Will it work if we call the method as in: `print_string(6)` ?

```
1      >>> print_string(6)
2      6
3      6
4      6
5      6
6      6
```

`6` is passed as the first parameter. `6` is matched to `str`, and the method prints `6` the default number of times, which is `5`.

to default to `"Hello World"`, and print it `6` times.

You can do this in Python by using **named parameters**. During the method call, you can specify `no_of_times = 6`. `no_of_times` is a named parameter.

| There is no provision of doing something like this, in other languages like Java.

Call it as `print_string(no_of_times=6)` :

```
1      >>> print_string(no_of_times=6)
2      Hello World
3      Hello World
4      Hello World
5      Hello World
6      Hello World
7      Hello World
```

`str` gets a default value, and "Hello World" is printed 6 times.

Named parameters are very useful, when a method has a number of parameters, and you would want to make it very clear which parameter you're passing a value for.

Let's call `print_string(7, 8)`. what happens?

```
1      >>> print_string(7, 8)
2      7
3      7
4      7
5      7
6      7
7      7
8      7
9      7
```

You would see that 7 is printed 8 times.

Since `print()` method is quite flexible, you can pass a number as the first argument. You can even pass a `float`.

```
1      >>> print_string(7.5, 8)
2      7.5
3      7.5
4      7.5
5      7.5
6      7.5
7      7.5
8      7.5
9      7.5
```

What would be the result of this - `print_string(7.5, "eight")` ?

```
1      >>> print_string(7.5, "eight")
2      Traceback (most recent call last):
3          File "<stdin>", line 1, in <module>
4          File "<stdin>", line 2, in print_string
5      TypeError: must be str, not int
```

Note how `no_of_times` is used inside the method... as an argument to `range()`. `range()` only accepts integers, nothing else. When you run the code with `print_string(7.5, "eight")`, we get an error.

It says: `TypeError: ``no_of_times`` must be ``int``, not string`.

A simple rule of thumb is, if you have a parameter, you can pass any type of data to it. That could be an integer, a floating point value a string, or a boolean value. The Python language does not check for the type of a parameter. However, Python will throw an error if the function which is using that parameter, expects it to be of a specific type. The `range()` function expects that the `no_of_times` is an integer value.

Snippet-02:

The last thing which we would be looking at, is method naming conventions. We named our methods in a consistent way: `print_string`, `print_multiplication_table`, and the like.

This is exactly the format which most Python developers use, to name their methods.

Convention is to use underscore to separate words in a name.

However, there are a few rules for naming a method: One of the important rules is also related to variable names. We observed that a variable name cannot start with a number.

```
1      >>> def 1_print():
2          File "<stdin>", line 1
3              def 1_print():
4                  ^
5      SyntaxError: invalid token
```

Similarly, `1_print` will not be accepted as a method name.

- You can start a name with an alphabet, or with an underscore.
- From the second character onward, you are allowed to use numeric symbols.

Methods and variables cannot be named using Python keywords.

Now, what is a keyword? For example, when we talked about `for` loop, as in:

```
```for i in range(1, 11): print(i)```...
```

- `for` is a keyword
- `in` is a keyword
- `def` is a keyword.

Later we will look at a few other keywords, such as `while`, `return`, `if`, `else`, `elif`, and many more.

```
1 >>> def def():
2 File "<stdin>", line 1
3 def def():
4 ^
5 SyntaxError: invalid syntax
6 >>> def in():
7 File "<stdin>", line 1
8 def in():
9 ^
10 SyntaxError: invalid syntax
11 >>> def for():
12 File "<stdin>", line 1
13 def for():
14 ^
15 SyntaxError: invalid syntax
```

## Summary

In this step, we:

- Were introduced to the concept of named parameters
- Explored the typical naming rules and conventions for methods in Python
- Observed that reserved keywords cannot be used to name variables or methods

## Step 09: Methods - Return Values

Let's try and understand the importance of return values from a method. We will learn how to return a value from a method.

### Snippet-01:

Let's name our method as `product_of_two_numbers()`, and let's have parameters `a` and `b` that it accepts:

```
1 >>> def product_of_two_numbers(a,b):
2 ... print(a * b)
3 ...
4 >>> product_of_two_numbers(1,2)
5 2
```

Can we take the product of these two numbers into a variable, and use it in other code, in the same program?

Suppose we say a `product = product_of_two_numbers(1,2)`, is this allowed?

Let's run this code, and see what's stored in `product`.

```
1 >>> product = product_of_two_numbers(1,2)
2 2
3 >>> product
```

It's empty.

The `product_of_two_numbers()` method is not really returning anything back, to be used elsewhere.

Have a look at some of the built-in Python functions, such as `max()` for example.

```
1 >>> max(1,2,3)
2 3
3 >>> max(1,2,3,4)
4 4
5 >>> maximum = max(1,2,3,4)
6 >>> maximum
7 4
8 >>> maximum * 5
9 20
```

If I call `max()` with four parameters, as in `maximum = max(1,2,3,4)`, the value `4` gets stored in `maximum`.

Later on in the code that follows, we can say `maximum * 5`, or we can print the value of `maximum`, or a similar calculation. This gives our programs a lot more flexibility.

So instead of just printing `a*b`, if this function could return a value, that would be quite useful.

```
1 >>> def product_of_two_numbers(a,b):
2 ... product = a * b;
3 ... return product
4 ...
5 >>> product_of_two_numbers(2,3)
6 6
```

We are creating a variable `product` and doing a `return product`.

Lets run `product_result = product_of_two_numbers(2, 3)`

```
1 >>> product_result = product_of_two_numbers(2,3)
2 >>> product_result
3 6
4 >>> product_result * 10
5 60
```

You can see how simple it is to return values from a method!

## Summary

In this step, we:

- Learned how to return values from inside a method
- Observed how we can store the values returned by a method call

## Step 10: Programming Exercise PE-MD-02

In this step let's look at a couple of exercises about returning values from methods.

### Exercises

1. Write a method to return the sum of three integers.
2. Write a method which takes as input two integers, representing two angles of a triangle, and computes the third angle.

Hint: The sum of the angles in a triangle is 180 degrees. So if I am passing 50 and 50, 50 plus 50 is 100. So some of three angles should be 180, so the third angle will be 180 - 100, which is 80.

### Solution 1

```
1 >>>def sum_of_three_numbers(a, b, c):
2 ... sum = a + b + c
3 ... return sum
4 ...
5
6 >>> sum_of_three_numbers(1,2,3)
7 6
8 >>> something = sum_of_three_numbers(1,2,3)
9 >>> something * 5
10 30
```

The shorter way of doing that would have been to have a temporary variable called instead of sum. We could directly return a + b + c.

```
1 >>> def sum_of_three_numbers(a, b, c):
2 ... return a + b + c
3 ...
4 >>> something = sum_of_three_numbers(1,2,3)
5 >>> something * 5
6 30
```

In methods, you can use `return expression` as well. That `expression` gets evaluated, and the value gets returned back. You'd see that the result remains the same.

## Solution 2

The second is to write a method to take two integers, representing two angles of a triangle, and compute the third one.

```
1 >>> def calculate_third_angle(first, second) :
2 ... return 180 - (first + second)
3 ...
4 >>> calculate_third_angle(50, 20)
5 110
```

In your programming career, you would be writing a number of methods. It's very important that you are comfortable doing so. Most of the methods that you write would return values back.

That's the reason why we're creating a lot of examples involving method calls.

## Summary

In this step, we:

- Looked at a couple of exercises related to returning values from methods
- Observed that returning expressions avoids creating unnecessary variables, and shortens method definitions

## Chapter 04 - Introduction To Python Platform

Until now we had been using Python Shell to execute all our code.

In the real world, we'll be writing Python code in a variety of scripts. Before we would go into an IDE and use the IDE to write the script, we thought it would be useful for us to understand how you can write Python code without the benefit of an IDE.

This would also help us understand the Python environment, in-depth.

In the next few steps, we'll be looking at how to create simple Python scripts, using any text editor of your choice. Use Notepad, Notepad++, Editpad, or whichever text editing software you are comfortable with. We'll see what's involved in executing the program, and what's happening in the background.

Here are a few videos you might want to look at.

- Writing and Executing your First Python Script
- Understanding Python Virtual Machine and bytecode

### **Step 01 - Writing and Executing Python Shell Programs**

Here's a recommended video to watch - Writing and Executing your First Python Script

Let's get started with creating a simple script file.

We want to type in a simple Python script, or a piece of Python code, such as

```
print("Hello world")
```

We'll save this into any folder on our hard disk, with a name 'first.py' .

**first.py**

```
print("Hello world")
```

The '.py' is not really mandatory, but typically all python files end with a '.py' extension.

Here's how you can run it:

- Launch your terminal, or command prompt
- 'cd' to the folder where this python script file is saved

- execute the command `python first.py`

You will see that `Hello World` will be printed.

If you are familiar with other programming languages, you would need a class, need to put the code in that class, and similar stuff.

While Python supports Object Oriented Programming, is not mandatory to create a class.

It's almost as if you're typing commands, starting from the line one! That's why we call it a python script.

## Summary

In this small step, we tried to create a simple python script, and we ran it from the command line. All we needed to do, was use the same command we use to launch up the python shell, and followed it up with a name of the file. We created a file called `first.py`, executed that, and were able to see the output on the console.

As an exercise, try and add a few more methods and try to run those methods as well, as part of this script.

## Step 02 - Python virtual machine and bytecode

In this step, let's try and understand what's happening in the background.

We wrote a simple piece of code using a text editor. We created a file named `first.py`, and all we did was: `python3 first.py`. If you look at other languages like Java for example, there is a separate compilation phase and then an execution phase. But with Python, just this command does both compilation and execution.

We saw that, as soon as we make a change and we run `python3 first.py`, the change is compiled and executed as well!

In Python, there is an intermediate format called **Python byte code**. Code is first compiled to bytecode, and then executed on the **Python virtual machine**.

When we installed Python, we installed both the python compiler and interpreter, as well as the virtual machine.

In Python, `bytecode` is not standardized. Different implementations of Python have different byte code. There are about 80 Python implementations, like CPython and Jython.

- CPython is a Python implementation in C language.
- Jython is a Python implementation in Java language. The bytecode which Jython uses is actually Java bytecode, and you can run it on the Java virtual machine.

Python leaves a lot of flexibility to the implementations of Python. They have the flexibility to choose the bytecode, and to choose the virtual machine that is compatible. The bytecode is tied to the specific virtual machine you are using. Therefore, if you're using CPython to compile the bytecode, you'll not be able to use Jython to run it.

You should make sure, that whatever implementation you are using to compile, is the same one you're using to run the code as well.

## Summary

A lot of this sounds like boring theory. Don't worry about it. As a beginner, this might not be very important for you right now.

It's very important for you to understand the process. What's happening is you were writing Python code, and when you ran the command `python3 first.py`, it is both compiled and executed. An intermediate format called bytecode is created, which is not really standardized in Python. The bytecode is executed in a Python virtual machine.

The idea behind this quick section, is to give you a little bit of background on what's happening behind the scenes. I'll see you in the next section. Until then, bye-bye!

## Chapter 05 - Introduction To VSCode

Let's start using the IDE VSCode to write our Python Code

Here are recommended videos to watch

- Installing VSCode
- Write and Execute a Python File with VSCode
- Write Your First Python Program with VSCode

### Step 01 - Installing and Introduction to VSCode

In this quick step, we'll help you install VSCode.

Here's the video guide for this step

- Installing VSCode

Go to Google and type in "VSCode Community Edition Download". Click the link which comes up first: <https://www.jetbrains.com/VSCode/download>.

You'll go to a page where you can choose the operating system: whether you are on Windows, Mac, or Linux.

Once you choose that, you can download the appropriate community version.

On the right hand side, you'll see a community version, and you can click the download link, to start the download.

If you are having a problem, you can also use the direct link to download.

Once you download VSCode, all you need to do is double-click the package which is downloaded. Follow the instructions, and you can continue with the defaults, until you completely install VSCode.

When you launch VSCode for the first time, it should ask you for a theme, where you can choose the default.

You're all set to go ahead with the next step in the course.

VSCode is an awesome IDE, and I'm sure you learn a lot about it.

## **Step 02 - Write and Execute a Python File with VSCode**

In this step, let's launch up the VSCode IDE, and create our first Python project with a Python script. We want to be able to launch a Python script by the end of this step.

Here's the video guide for this step

- Write and Execute a Python File with VSCode

Launch the VSCode IDE. You'll see that it takes a little while to launch the first time, and then brings up a welcome screen.

We would want to create a number of Python files. All these files will be in a project. You can think of our project as a collection of Python scripts, or modules.

To get started, let's create a new project by clicking 'create new project'. Let's name it - '01-first-python-project'.

Right now there are no files in the project.

Let's create our first Python file, using the IDE.

The way you can do that is by saying 'right-click' -> 'new' -> 'Python file', and then we'll give this a name of 'hello\_world', and click OK.

Now you can go ahead and write your first Python program. Let's write some simple code, like `print("Hello World")`, and save it.

You can do a right-click here, and say 'Run hello\_world'.

A small window comes up below, which shows the output. It says `'Hello World'`.

### **Step 03 - Exercise - Write Multiplication Table Method with VSCode**

Let's start with a simple exercise. We created the multiplication table method in the Python Shell. What we do now, is we'll create the same thing but in a Python file of its own.

Here's the video guide for this step:

- Write Your First Python Program with VSCode

## **Chapter 06 - Introducing Data Types and Conditionals**

Welcome to this section, where we will talk about numeric data types, and conditional program execution. After looking at the numeric and boolean data types, we will turn our attention to executing code, based on logical conditions.

### **Step 01: Numeric Data Types**

In previous sections, we created variables of this kind: `number = 5`, `value = 2.5`, etc. The `5` here is an integer, and integers represent numbers, such as `1`, `2`, `6`, `-1` and `-2`. In Python, the `class` for this particular data type is `int`.

If you write code like `type(5)`, you'd get `'int'` as the output.

In Python, there are no primitive types. What does that mean? Every value that you see in a Python program, is an object, an instance of some `class`.

In later sections, We'll understand what is a `class`, and what is an object or an instance. For now, the most important thing for you to remember, is that behind every value, there is a `class`.

### Snippet-01:

Let's look at `2.5`, which is a floating point value.

If you go ahead and do `type(2.5)`, what would you see? You would see it's of type `'float'`.

```
1 >>> type(2.5)
2 <class 'float'>
3 >>> type(2.55)
4 <class 'float'>
```

When you perform a division operation between two integers, there is a chance that the result of the operation is a `float`. If you do `5/2`, the result is `2.5`. If we were to do `4/2`, even then it's of type `float`.

```
1 >>> type(5/2)
2 <class 'float'>
3 >>> type(4/2)
4 <class 'float'>
5 >>> 4/2
6 2.0
7 >>> 1 + 2
8 3
```

All the operations we looked at until now, can also be performed on floating point values.

```
1 >>> value1 = 4.5
2 >>> value2 = 3.2
3 >>> value1 + value2
4 7.7
5 >>> value1 - value2
6 1.299999999999998
7 >>> value1 / value2
8 1.40625
9 >>> value1 % value2
10 1.299999999999998
```

`value1 - value2` returns `1.299999999999998`. Why?

Floating point numbers don't really represent accurate values. That's one of the things you need to always keep in mind.

Typically, if you're doing any highly sensitive financial calculations, don't use `float`s to represent your values. Instead, use `Decimal`. More about it later.

Operations can also be performed between `int` and `float`.

```
1 >>> i + value1
2 14.5
3 >>> i - value1
4 5.5
5 >>> i / value1
6 2.222222222222223
7 >>>
```

Result of an operation between a `int` and a `float`, is always a `float`.

## Summary

In this step, we:

- Looked at the two basic numeric types: `int` and `float`.

- Saw the basic operations you can do among `int`s, among `float`s, and also between `int`s and `float`s.

## Step 02: Programming Exercise PE-DT-01

In this step, let's do a simple exercise with numeric values.

### Exercises

1. You need to create a method called `simple_interest`, and pass three parameters: `principal`, `interest` and `duration` (in years). You also want to calculate the amount after the specific duration, and return it back. Call this method with a few example values.

For example, if you want to call `simple_interest` with `10000`, with an interest of `5` percent, for a duration of `5` years, the correct answer would be as follows: `10000` is the principal. In addition to `10000`, you get the interest. The interest for one year is `10000 * 0.05`, as the interest figure is in percentage. So that's `500` a year, into `5` which is `2500`. The result would be `12500`, and this value should be printed.

### Solution 1

```
1 def calculate_simple_interest(principal, interest, duration) :
2 return principal * (1 + interest * 0.01 * duration)
3
4 print(calculate_simple_interest(10000,5,5))
```

### Summary

In this step, we:

- Wrote a very simple method to do a simple interest calculation

## Step 03: Puzzles On Numeric Types

In this section, we are looking at numeric types. In this specific step, we would be looking at a few puzzles related to values of these types.

### Snippet-01:

Let's create a simple variable `i = 1`. `i = i + 1`. What would be the value of `i` after that?

```
1 >>> i = 1
2 >>> i = i + 1
3 >>> i
4 2
```

It would be `2`. There is a shortcut way of doing the same thing, by using the `+=` operator.

```
1 >>> i += 1
2 >>> i
3 3
```

Typically in other programming languages, you can do something of this kind: `i++`. There is no provision in Python to use increment operators like `++`, in either prefix or suffix mode, like `++i`, or `i++`.

```
1 >>> i++
2 File "<stdin>", line 1
3 i++
4 ^
5 SyntaxError: invalid syntax
6 >>> ++i
7 3
```

Let's look at compound assignments.

```
1 >>> i += 1
2 >>> i
3 4
4 >>> i -= 1
5 >>> i
6 3
7 >>> i /= 1
8 >>> i *= 2
9 >>> i
10 6.0
```

What you see here, is Dynamic Typing in Python. The type of a variable can change during the lifetime of the program.

```
1 >>> i = 2
2 >>> type(i)
3 <type 'int'>
4 >>> i = i/2.0
5 >>> type(i)
6 <type 'float'>
```

Let's create a couple more numbers. `number1 = 5` and `number2 = 2`. What could be the result of `number1 / number2`? You know it, it's `2.5`.

`number1 // number2` truncates the value of `2.5`, to `2`.

```
1 >>> number1//number2
2 2
```

If you can do `number1 // number2`, can you also do this: `number1 //= number2`?

```
1 >>> number1 //= 2
2 >>> number1
3 2
```

`5 ** 3` is `5` 'to the power of' `3`, which is `5 * 5 * 5`, or `125`.

```
1 >>> 5 ** 3
2 125
3 >>> pow(5,3)
4 125
```

This can also be achieved by invoking `pow(5, 3)` . We have an operator, as well as a method at our disposal.

The last thing we will look at, are type conversion functions.

If you need to convert an `int` value to a `float`, or a `float` to an `int` .

```
1 >>> int(5.6)
2 5
```

What if you want to round a value? `5.6` is nearer to `6` than `5` . You can use a function called `round()` , and here, `round(5.6)` gives the correct result `6` .

```
1 >>> round(5.6)
2 6
3 >>> round(5.4)
4 5
5 >>> round(5.5)
6 6
```

`round()` can also allows you to specify number of decimals in the result.

```
1 >>> round(5.67, 1)
2 5.7
3 >>> round(5.678, 2)
4 5.68
```

You can also convert `int` to `float` , by using the function `float()` .

```
1 >>> float(5)
2 5.0
```

## Summary

In this step, we:

- Looked at a few corner cases related to your numeric types.
- Examined the different operators available for use with values of numeric types
- Learned about the usage of type conversion functions

## Step 04: Introducing Boolean Type

We will now shift our attention to the `bool` data type.

A boolean value is something which can be either “true” or “false”.

### Snippet-01:

In Python, “true” is represented by `True`, and “false” by `False`. It’s important to remember that it’s `True` with a capital ‘T’, and `False` with a capital ‘F’.

```
1 >>> True
2 True
3 >>> False
4 False
5 >>> true
6 Traceback (most recent call last):
7 File "<stdin>", line 1, in <module>
8 NameError: name 'true' is not defined
9 >>> false
10 Traceback (most recent call last):
11 File "<stdin>", line 1, in <module>
12 NameError: name 'false' is not defined
```

The boolean variable `is_even` indicates whether a number is even or not.

```
1 >>> is_even = True
2 >>> is_odd = False
```

Let’s create a variable `i = 10`. We want to find out if `i > 15`. What do you think is the result? `False`.

```
1 >>> i = 10
2 >>> i > 15
3 False
4 >>> i < 15
5 True
```

In general, boolean values can represent the result of logical conditions.

Let's look at other operations that can result in `bool` values. We looked at `>` and `<`. Another operation which you can perform, is `>=`.

```
1 >>> i >= 15
2 False
3 >>> i >= 10
4 True
5 >>> i > 10
6 False
7 >>> i <= 10
8 True
9 >>> i < 10
10 False
```

`==` is the comparison operator. We are only comparing the value of `i` against `10`, not changing its value.

```
1 >>> i == 10
2 True
3 >>> i == 11
4 False
```

## Summary

In this step, we:

- Were introduced to the `bool` data type
- Learned that `bool` variables are useful handy while testing logical conditions

## Step 05: Introducing Conditionals

In this step, let's look at `if` statement.

Sometimes you need to execute code only when certain conditions are true. You can use a `if` condition, which is the simplest conditional in Python. Let's look at an example.

### Snippet-01:

Let's say `i` has a value of `5`. You want to print something, only if `i` has a value greater than `3`. How do you do that?

```
1 >>> i = 5
2 >>> if i>3:
3 ... print(f"{i} is greater than 3")
4 ...
5 5 is greater than 3
```

The syntax of the `if` is very simple: `if` followed by a condition; with the condition you want to check. It looks like: `if i>3: ...` You need to indent the body of the `if` with `<SPACE>`s as usual.

Let's say `i` has a value of `2`. What would happen if we execute the same code again?

```
1 >>> i = 2
2 >>> if i>3:
3 ... print(f"{i} is greater than 3")
4 ...
```

You would see that nothing is printed to the console. Based on the value of `i`, either the statement is executed, or it's not. That's what an `if` helps us to do.

The way you can think about an `if`, is the body of code under the `if` is executed only when this condition is `True`. If this condition is not `True`, that code is not executed at all.

```
1 >>> if(False):
2 ... print("False")
3 ...
4 >>> if(True):
5 ... print("True")
6 ...
7 True
```

Let's take two different numbers, say `a = 5`, and `b = 7`. We want to compare them, and predict if `a` is greater than `b`.

```
1 >>> a = 5
2 >>> b = 7
3 >>> if(a>b):
4 ... print("a is greater than b")
5 ...
6
7 >>> a = 9
8 >>> if(a>b):
9 ... print("a is greater than b")
10 ...
11 a is greater than b
```

## Summary

In this step, we:

- Were introduced to the `if` statement, the simplest Python conditional
- Understood how an `if` helps in implementing conditional program logic

## Step 06: Classroom Exercise CE-DT-01

In this step, let's look at a couple of exercises with the `if` statement.

### Snippet-01:

Let's say we define four variables: `a = 1`, `b = 2`, `c = 3` and `d = 5`. we want to find out, if `a + b` is greater than `c + d`.

```
1 >>> a = 1
2 >>> b = 2
3 >>> c = 3
4 >>> d = 5
5 >>> if a+b > c+d :
6 ... print("a+b > c +d")
7 ...
8 >>> a = 9
9 >>> if a+b > c+d :
10 ... print("a+b > c +d")
11 ...
12 a+b > c +d
```

Let's say we are given three values meant to be the angles of a triangle. Their values are

`angle1 = 30` , `angle2 = 20` and `angle3 = 60` . You want to find out if these three angles actually form a valid triangle. You know that the sum of the angles of a triangle is always `180` degrees.

```
1 >>> angle1 = 30
2 >>> angle2 = 20
3 >>> angle3 = 60
4 >>> if(angle1 + angle2 + angle3 == 180):
5 ... print("Valid Triangle")
6 ...
7 >>> angle2 = 90
8 >>> if(angle1 + angle2 + angle3 == 180):
9 ... print("Valid Triangle")
10 ...
11 Valid Triangle
```

The last exercise is to check if a number is even or not.

Hint L you need to use one of the operators we talked about earlier. That's right, use the modulo operator `%` .

```
1 >>> i = 2
2 >>> if(i%2==0):
3 ... print("i is even")
4 ...
5 i is even
6
```

```
7 >>> i = 3
8 >>> if(i%2==0):
9 ... print("i is even")
10 ...
```

## Summary

In this step, we:

- Looked at a few exercises related to the if statement, for writing and testing conditions.

## Step 07 - Logical Operators - and or not

In this step, let's look at the different operators that can be used on `bool` values. These operators are called logical operators - `and` , `or` , `not` and `^` (xor).

Let's say we have a value `True` , and the other `False` , and we want to play around with them.

Logical operator `and` returns true only when both operands are `True` .

```
1 >>> True and False
2 False
3 >>> True and True
4 True
5 >>> True and False
6 False
7 >>> False and True
8 False
9 >>> False and False
```

Logical operator `or` returns true when atleast one of the operands is `True` .

```
1 False
2 >>> True or False
3 True
4 >>> False or True
5 True
6 >>> True or True
7 True
8 >>> False or False
```

Logical operator `not` returns negation.

```
1 False
2 >>> not True
3 False
4 >>> not(True)
5 False
6 >>> not False
7 True
8 >>> not(False)
9 True
```

The XOR operation, denoted by the `^` operator, is `True` when operands have different boolean values.

```
1 >>> True ^ True
2 False
3 >>> True ^ False
4 True
5 >>> False ^ True
6 True
7 >>> False ^ False
8 False
```

## Summary

In this step, we:

- Looked at the logical operators that act on boolean values, such as `and`, `or`, `not` and `^`
- Explored each of these operators, finding out when they return `True`, and when `False`.

## Step 08: Puzzles On Logical Operators

In this step, Let's look at a few simple puzzles to look at the logical operators.

### Snippet-01:

Let's say `i` has a value of `10`, and `j` has a value of `15`. You want to find out if both `i` and `j` are even. How do you do that?

```
1 >>> i = 10
2 >>> j = 15
3 >>> if i%2==0 and j%2==0:
4 ... print("i and j are even")
5 ...
6 >>> j = 14
7 >>> if i%2==0 and j%2==0:
8 ... print("i and j are even")
9 ...
10 i and j are even
11
12 >>> if i%2==0 or j%2==0:
13 ...
14 File "<stdin>", line 2
15 ^
16 IndentationError: expected an indented block
17 >>> if i%2==0 or j%2==0:
18 ... print("atleast one of i and j are even")
19 ...
20 atleast one of i and j are even
```

If we want to find out if at least one of `i` and `j` is even, we can use the `or` operator.

```
1 >>> i = 15
2 >>> j
3 14
4 >>> if i%2==0 or j%2==0:
5 ... print("atleast one of i and j are even")
6 ...
7 atleast one of i and j are even
8 >>> j = 23
9 >>> if i%2==0 or j%2==0:
10 ... print("atleast one of i and j are even")
11 ...
12 >>> i
13 15
```

Now try and guess the value of this. `if(True ^ False): print("Message")`

```
1 >>> if(True ^ False):
2 ... print("This will Print")
3 ...
4 This will Print
5 >>> if(False ^ True):
6 ... print("This will Print")
7 ...
8 This will Print
9 >>> if(True ^ True):
10 ... print("This will Print")
11 ...
```

Xor operation using `^` - message will get printed if the operands are different.

What would happen if both of them are `True`? No message is printed.

So you would use `^` in situations, where you'd want one of the operands to be `True`, and the other to be `False`.

Let's say, `x = 5`, and you want to check `if not x == 6: print("This")`. What will be the result of running this code?

```
1 >>> x = 5
2 >>> if not x == 6:
3 ... print("This")
4 ...
5 This
6 >>> x = 6
7 >>> if not x == 6:
8 ... print("This")
9 ...
```

Actually, there is a shortcut for such a condition: `if x != 6 : print("This")`.

```
1 >>> if x!=6:
2 ... print("This")
3 ...
4 >>> x=5
5 >>> if x!=6:
6 ... print("This")
7 ...
```

8 This

`int()` is a conversion function, which when given say a `float` value, returns an `int` value.  
Consider `int(True)`, what would happen?

```
1 >>> int(True)
2 1
3 >>> int(False)
4 0
```

`int(True)` returns 1. `int(False)` returns 0.

```
1 >>> x = -6
2 >>> if x:
3 ... print("something")
4 ...
5 something
```

One of the most interesting facts about boolean stuff, is anything which is non-zero, is considered to be `True`.

`0` is the only integer value which is considered to be `False`.

```
1 >>> bool(6)
2 True
3 >>> bool(-6)
4 True
5 >>> bool(0)
6 False
7 >>>
```

So, if I have a value of `x = -6`, and execute `if x: print("something")` what do you think will happen?

"something" will be printed.

You can use the function `bool()`, to convert `int` to a `bool` value.

- `bool(6)` returns `True`
- `bool(-6)` returns `True`
- `bool(0)` returns `False`.

Except for `bool(0)`, all the other results would be `True`.

## Summary

In this step, we:

- Looked at a few puzzles related to the logical operators
- Looked at conversion functions such as `bool()` and `int()` to convert between boolean and integer data

## Step 09:

In this step, let's look at two other important components of an `if` statement: `else` and `elif`. Let's start with `else`.

### Snippet-01:

Consider a scenario where `i` has a value of `2`. Let's try to print a message "`i` is even" if `i` is an even number. Otherwise, print "`i` is odd".

Earlier we wrote code along these lines: `if i % 2 == 0 : print("i is even")`. However if this condition is not `True`, we would want to `print("i is odd")`. How do we accomplish that?

```
1 >>> i = 2
2 >>> if i%2 == 0:
3 ... print("i is even");
4 ... else:
5 ... print("i is odd");
6 ...
7 i is even
```

An `else` clause provides an alternative code body to execute, if the `if` condition is `False`.

```
1 >>> i = 3
2 >>> if i%2 == 0:
3 ... print("i is even");
4 ... else:
5 ... print("i is odd");
6 ...
7 i is odd
```

Let's look at `elif`.

We want to do something if `i` has value of `3`, and something totally different if `i` has a value of `4`.

In short, we want to specify 2 alternatives to the `if` condition. How can that be done?

```
1 >>> if i==1:
2 ... print("i is 1")
3 ... elif i==2:
4 ... print("i is 2")
5 ... else:
6 ... print("i is not 1 or 2")
7 ...
8 i is not 1 or 2
9 >>>
```

That's where the `elif` clause comes into the picture. The code in `elif` is executed if the previous conditions are false and the current `elif` condition is true.

## Summary

In this step, we:

- Looked at two important components of the `if` statement: `else` and `elif`.
- Understood that the `elif` clauses and the final `else` clause provide alternative conditions to check, when earlier if conditions are true.

## Step 10: Classroom Exercise CE-DT-02

In this step, let's do a simple exercise with `if`, `else` and `elif`.

Before getting to the exercise, let's try and learn how to get console input from the user.

Until now, we had been hard-coding all the data we were to use. Let's make that part more dynamic now.

### Snippet-01:

How do we get input from the user? We want to get input from the console, and assign it to a variable. The way we can do that, is by statement `value = input()`

```
1 value = input("Enter a Value: ")
2 print("you entered ", value)
```

We can call the `input()` method with a text 'prompt', such as `"Enter A Value: "`. What we can initially do here, is print the value which was entered, back to the console, by `print("you entered ", integer_value)`.

An interesting point to explore here, is the type of data input at the console.

Let's do a `print(type(value))`.

```
1 value = input("Enter a Value: ")
2 print("you entered ", value)
3 print(type(value))
```

Input a value of `Test`. It has a class of `str`.

Let's run it again to see other possibilities. This time, let's enter a numeric value, say `12`. what would happen?

We again get `str`.

We want to get an integer value from the input. How can we do it?

`int()` function converts string to int. Let's use it.

```
1 value = input("Enter a Value: ")
2 integer_value = int(value)
3 print("you entered ", integer_value)
4 print(type(integer_value))
```

Let's run our code once again.

"Enter A Value: " is prompted, and we enter 15 . And now, of it says "You entered 15" , and the type it indicates to us, is `int` .

## Design a menu

- Ask the User for input:
  - Enter two numbers
  - Choose the Option:
    - 1 - Add
    - 2 - Subtract
    - 3 - Multiply
    - 4 - Divide
- Perform the Operation
- Publish the Result

Let's design a menu, and then ask the user for input.

We have codes for each of the operations : add is 1 , subtract is 2 , divide is 3 , and multiply is 4 .

In the first version of the program let's get all the inputs and print them out.

## Solution

The first version of the program is simple to write

```
1 number1 = int(input("Enter Number1: "))
2 number2 = int(input("Enter Number2: "))
3 print(f"You entered {number1}")
4 print(f"You entered {number2}")
5 print(number1 + number2)
6 print("\n\n1 - Add")
7 print("2 - Subtract")
8 print("3 - Divide")
9 print("4 - Multiply")
10 print("5 - Exit")
11 choice = int(input("Choose Operation: "))
12 print(choice)
```

We will continue this exercise to complete it, in the next step.

## Summary

In this step, we:

- Looked at the in-built `input()` function that can read console input
- Learned that `input()` always returns what the user enters, as a string
- We can convert the string from `input()`, to the data type we expect by invoking conversion functions

## Step 11: Continued - Classroom Exercise CE-DT-02

### Exercises

In the previous step, we got the input from the user. Let's continue the exercise in this step. We want to write an if condition.

### Solution (Continued)

Extending the solution is easy. Write appropriate `if`, `elif` and `else` conditions.

```
1 number1 = int(input("Enter Number1: "))
2 number2 = int(input("Enter Number2: "))
3
4 print("\n\n1 - Add")
5 print("2 - Subtract")
6 print("3 - Divide")
7 print("4 - Multiply")
```

```
8
9 choice = int(input("Choose Operation: "))
10
11 # print(number1 + number2)
12 # print(choice)
13 if choice==1:
14 result = number1 + number2
15 elif choice==2:
16 result = number1 - number2
17 elif choice==3:
18 result = number1 / number2
19 elif choice==4:
20 result = number1 * number2
21 else:
22 result = "Invalid Choice"
23
24 print(result)
```

We added the following code to account for invalid input.

```
1 else:
2 result = "Invalid Choice"
```

## Summary

In this step, we:

- Augmented the Menu Exercise to get all the input from the console, and compute a value from them
- Corrected the logic to handle incorrect input

## Step 12: Puzzles On Conditionals

In this step, let's look at a few puzzles related to these `if`, `elif` and `else` clauses.

### Puzzle-01

Let's start with the first puzzle. Guess the output.

```
1 k = 15
```

```
2 if (k > 20):
3 print(1)
4 elif (k > 10):
5 print(2)
6 elif (k < 20):
7 print(3)
8 else:
9 print(4)
```

When we run it, you can see that the output is 2 .

k has a value of 15 , is it greater than 20 ? No! Execution goes to the elif , is k greater than 10 ? Yes. It prints 2 and goes out of the complete if - else block.

Inside the if conditional, the if , elif and else clauses are all independent ones. Only one matching block is ever executed.

## Puzzle-02

What do you think would be the output of this particular piece of code?

```
1 l = 15
2 if (l < 20):
3 print("l<20")
4 if (l > 20):
5 print("l>20")
6 else:
7 print("Who am I?")
```

Note that there are two totally different if conditions in here : if l < 20: ... immediately followed by if l > 20: ... else: ....

The first if is true. l<20 is printed.

The second if is a separate statement. The condition is false. So. else gets executed. Therefore, "who am I" gets printed.

## Puzzle-03

Let's run this code.

```
1 m = 15
2 if m>20:
3 if m<20:
4 print("m>20")
5 else:
6 print("Who am I?")
```

You can see that nothing is printed.

The most important thing to focus on here, is indentation.

The second `if` block is executed only if the first `if` is true.

#### Puzzle-04

What would be the output?

```
1 number = 5
2 if number < 0:
3 number = number + 10
4 number = number + 5
5 print(number)
```

`10` is printed.

The most important thing to focus on here, is indentation.

Only `number = number + 10` is part of `if` block. It is not executed because the condition is false.

`number = number + 5` is not part of `if`. So, it gets executed.

Let's add a couple of spaces before `number = number + 5`.

What would be the output?

```
1 number = 5
2 if number < 0:
3 number = number + 10
4 number = number + 5
5 print(number)
```

5 is printed.

Both the statements `number = number + 10` and `number = number + 5` are part of `if` block. They are not executed because the condition is false.

## Summary

In this step, we:

- Looked at a few puzzles related to `if`, `elif` and `else`
- Explored the importance of indentation and the different condition clauses inside an `if` statement

## Step 01: The Python Type To Denote Text

Let's start looking at another important data type in Python, that's used to represent strings. Not surprisingly, it is in fact named `str`!

Let's look at valid string representations.

```
1 >>> message = "Hello World"
2 >>> message = 'Hello World'
3 >>> message = 'Hello World'
4 File "<stdin>", line 1
5 message = 'Hello World' ^
6 SyntaxError: EOL while scanning string literal
```

In Python, you can use either `'''` or `'''` to delimit string values.

`type()` method can be used to find type of a variable.

```
1 >>> message = "Hello World"
2 >>> type(message)
3 <class 'str'>
```

The `str` class provides a lot of utility methods.

```
1 >>> message.upper()
2 'HELLO WORLD'
3 >>> message.lower()
4 'hello world'
5 >>> message = "hello"
```

`message.capitalize()` does init caps. Only first character is changed to uppercase.

```
1 >>> "hello".capitalize()
2 'Hello'
3 >>> 'hello'.capitalize()
4 'Hello'
```

You can also run this directly - `'hello'.capitalize()`. Isn't that cool!

That's because each piece of text in python is an object of the `str` class, and we can directly call methods of that `class` on `str` objects.

Now let's shift our attention to methods, which gives us more information about the specific contents of a string.

- We want to find out if this string contains numeric values?
- Does it contain alphabets only?
- Does it contain alpha-numeric values?
- Is it lowercase?
- Is it uppercase?

To find if a piece of text contains only lower case alphabets.

```
1 >>> 'hello'.islower()
2 True
3 >>> 'Hello'.islower()
4 False
```

If the first letter is in uppercase, then `istitle()` will return a `True` value.

```
1 >>> 'Hello'.istitle()
2 True
3 >>> 'hello'.istitle()
4 False
```

To find if a piece of text contains only upper case alphabets.

```
1 >>> 'hello'.isupper()
2 False
3 >>> 'Hello'.isupper()
4 False
5 >>> 'HELLO'.isupper()
6 True
```

`isdigit()` checks if a string is a numeric value.

```
1 >>> '123'.isdigit()
2 True
3 >>> 'A23'.isdigit()
4 False
5 >>> '2 3'.isdigit()
6 False
7 >>> '23'.isdigit()
8 True
```

`isalpha()` checks if a string only contains alphabets.

```
1 >>> '23'.isalpha()
```

```
1 False
2 >>> '2A'.isalpha()
3 False
4 >>> 'ABC'.isalpha()
5 True
```

`isalnum()` checks if a string only contains alphabets and/or numerals.

```
1 >>> 'ABC123'.isalnum()
2 True
3 >>> 'ABC 123'.isalnum()
4 False
```

Lastly, we look at things which you can use, to check characters of a string.

`endswith` is self explanatory.

```
1 >>> 'Hello World'.endswith('World')
2 True
3 >>> 'Hello World'.endswith('ld')
4 True
5 >>> 'Hello World'.endswith('old')
6 False
7 >>> 'Hello World'.endswith('Wo')
8 False
```

`startswith` is self explanatory as well.

```
1 >>> 'Hello World'.startswith('Wo')
2 False
3 >>> 'Hello World'.startswith('He')
4 True
5 >>> 'Hello World'.startswith('Hello')
6 False
7 >>> 'Hello World'.startswith('Hello')
8 True
```

`find` method returns if a piece of text is present in another string. Returns the first match index.

```
1 >>> 'Hello World'.find('Hello')
2 0
3 >>> 'Hello World'.find('ello')
4 1
```

A value of `-1` is returned, if you're searching for something which is not present in the string.

If you are searching for `'Ello'` with a capital `'E'`, you'll not be able to find it. Search is case sensitive.

```
1 >>> 'Hello World'.find('Ello')
2 -1
3 >>> 'Hello World'.find('bello')
4 -1
5 >>> 'Hello World'.find('Ello')
6 -1
```

## Step 02: Type Conversion Puzzles

We'll now try and convert values from one type to another, and try and play around with them.

`str` converts boolean value to a text value.

```
1 >>> str(True)
2 'True'
```

All text value except for empty string represent True. So, `bool` returns True for everything except empty string.

```
1 >>> bool('True')
2 True
```

```
3 >>> bool('true')
4 True
5 >>> bool('tru')
6 True
7 >>> bool('false')
8 True
9 >>> bool('False')
10 True
11 >>> bool('')
12 False
```

Let's try and convert a few integer values to strings.

```
1 >>>str(123)
2 '123'
3 >>> str(12345)
4 '12345'
5 >>> str(12345.45678)
6 '12345.45678'
```

Let's do the reverse.

```
1 >>> int('45')
2 45
3 >>> int('45.56')
4 ValueError: invalid literal for int()
```

if we do `int('45.56')` , you can see that it throws an error. It says "I cannot convert this to an `int` , as `45.56` is an invalid integer".

You can also pass an additional parameter to `int` indicating the numeric system - 16 for Hexa decimal, 8 for Octal etc. Default is 10 - Decimal.

```
1 >>> int('45abc',16)
2 285372
3 >>> int('a',16)
4 10
5 >>> int('b',16)
```

```
6 11
7 >>> int('c',16)
8 12
9 >>> int('f',16)
10 15
11 >>> int('g',16)
12 ValueError: invalid literal for int() with base 16: 'g'
```

You can also convert string to float.

```
1 >>> float("34.43")
2 34.43
3 >>> float("34.43rer")
4 ValueError: could not convert string to float: '34.43rer'
```

## Summary

In this quick step, we looked at converting different types to strings, and converting strings to different types. So we looked at `int`, `bool` and `float` values, and we looked at how to convert them to string, and how to convert strings back to these specific types.

## Step 02: Strings Are Immutable

In this step, let's learn an important fact about strings in Python.

String values are immutable.

What does immutability mean, and why do we say strings are immutable?

Let's create a very simple string: `message = 'Hello'`, and we're saying `message.upper()`. But what does it do? It prints `'HELLO'`, with all characters in uppercase. Well, what would happen if you do `print(message)`? It says `'Hello'`.

```
1 >>> message = "Hello"
2 >>> message.upper()
3 'HELLO'
4 >>> message
5 'Hello'
```

You would see we tried change the content of message, but it has not changed.

When we execute `message.upper()`, a new string is created, and it is returned back. Original string remained unchanged. This is called immutability.

Once you define a string in Python, you'll not be able to change the value of it.

You can use - "OK. I can do something of this kind: `message = message.upper()`".

What would happen now?

Will the value of `message` get changed? It prints `'HELLO'`, with all caps.

Did the value of `message` change? Does this prove that strings are mutable?

The important thing you need to understand about all this stuff, is how objects are stored inside Python.

There are things called variables, and there are things called objects.

When we run `message = 'Hello'`

- We are creating one object of `str` class with a values `'Hello'`.
- We are creating one variable called `message`
- The location of `'Hello'` is stored into `message`

In Python, your variables are nothing but a name.

If location of `'Hello'` in memory is `A`, then the value stored in `message` is `A`. `message` is called a reference.

What happens with `message = message.upper()` ?

A new object is created with value `'HELLO'` at a different location `B`.

A reference of location `B` is stored into `message` variable.

Summary : The original value at location `A` has not changed and cannot be changed for `str` variables. Hence 'str' objects are immutable.

Variables are just names referring to a location. They don't really contain the value. Variables contain a reference to the location that contains the object.

### Step 03: Python Has No Separate Character Type

One of the things that surprises people new to Python, is that there is no character data type in Python.

Typically we have text data types in all the languages, don't we? 'Hello World' for example, is text data, and we stored it in `message`. This is called a string.

In other languages, you would have something to represent a single character symbol. For example in Java, you can have a `char` data type, to store a single character `ch`, in which '`h`' is one character. But in Python, there is no separate data type to store single characters.

For example, let's see how Python treats the first character of the following string `message`. The way you can access the first character of a string is by saying `message[0]`.

```
1 >>> message = "Hello World"
2 >>> message[0]
3 'H'
4 >>> type(message[0])
5 <class 'str'>
6 >>> type(message)
7 <class 'str'>
```

`type(message[0])` and `type(message)` print the same type `str`. No difference.

In Python, whether you're talking about a string, or you're talking about a single character symbol, they are all represented by the same `class`, `str`.

`message[100]` throws an `IndexError`.

```
1 >>> message[0]
2 'H'
3 >>> message[1]
4 'e'
5 >>> message[2]
```

```
6 'l'
7 >>> message[3]
8 'l'
9 >>> message[100]
10 IndexError: string index out of range
```

It says: "The given index is out of the range of the value of that specific string".

Let's say we would want to print all the characters in this string.

The way you could do that, is by saying: `for ch in message: print(ch)` .

## Summary

In this short step, we looked at the fact that there is no separate character class, or data type in Python. We also looked at how do we loop over a given string, and print all the characters present inside this string.

## Step 04: The `string` module

In this step, we will introduce you to the `string` module .

If we would want to use anything from a module in Python, you need to import that specific `module` into your program.

```
>>> import string
```

If you do a `string.` and press , it would show the different things which are part of the `string` module .

```
1 >>> string.
2 string.Formatter string.ascii_uppercase string.octdigits
3 string.Template(string.capwords(string.printable
4 string.ascii_letters string.digits string.punctuation
5 string.ascii_lowercase string.hexdigits string.whitespace
```

Let's explore some of these.

```
1 >>> string.ascii_letters
2 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
3 >>> string.ascii_lowercase
4 'abcdefghijklmnopqrstuvwxyz'
5 >>> string.ascii_uppercase
6 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
7 >>> string.digits
8 '0123456789'
9 >>> string.hexdigits
10 '0123456789abcdefABCDEF'
11 >>> string.punctuation
12 '!"#$%&\'()*+,-./:;<=>?@[\\\]^_`{|}~'
13 >>> string.ascii_letters
14 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

You have a set of printable characters, punctuation characters and a lot more.

You can check a text value against any of these

```
1 >>> 'a' in string.ascii_letters
2 True
3 >>> 'ab' in string.ascii_letters
4 True
5 >>> 'abc' in string.ascii_letters
6 True
```

`in` operation on a string, checks if a given string.

```
1 >>> '1' in '13579'
2 True
3 >>> '2' in '13579'
4 False
5 >>> '4' in '13579'
6 False
```

## Summary

In this step, we explored more exercises involving the `str` module of Python.

## Step 05: More Exercises With The `str` Module

Let's start with an Exercise - find if a specific character is a vowel or not.

```
1 >>> char = 'a'
2 >>> vowel_string = 'aeiouAEIOU'
3 >>> char in vowel_string
4 True
5 >>> char = 'b'
6 >>> char in vowel_string
7 False
```

The other thing you can do, is just have the capital vowels, or just the lowercase versions.

```
1 >>> vowel_string = 'AEIOU'
2 >>> char.upper() in vowel_string
3 False
4 >>> char = 'a'
5 >>> char.upper() in vowel_string
6 True
```

Now let's move on to the next one.

We want to find out and print all the capital alphabets, from `A` to `Z`.

There was a small clue at the start of the previous step, regarding importing the `string` module. We did the `string` module, and we saw that `string` module contained a number of things.

```
1 >>> string.ascii_uppercase
2 'ABCDEFGHIJKLMNPQRSTUVWXYZ'
3 >>> for char in string.ascii_uppercase:
4 ... print(char)
5 ...
6 A
7 B
```

```
8 C
9 D
10 E
11 F
12 G
13 H
14 I
15 J
16 K
17 L
18 M
19 N
20 O
21 P
22 Q
23 R
24 S
25 T
26 U
27 V
28 W
29 X
30 Y
31 Z
```

Try another easy exercise: print all the lower characters. Instead of `string.ascii_uppercase`, you have `string.ascii_lowercase`.

```
1 >>> for char in string.ascii_lowercase:
2 ... print(char)
3 ...
4 a
5 b
6 c
7 d
8 e
9 f
10 g
11 h
12 i
13 j
14 k
15 l
16 m
17 n
18 o
19 p
20 q
```

```
21 r
22 s
23 t
24 u
25 v
26 w
27 x
28 y
29 z
```

An even easier exercise, would be to print all the digits.

```
1 >>> for char in string.
2 string.Formatter(string.ascii_uppercase string.octdigits
3 string.Template(string.capwords(string.printable
4 string.ascii_letters string.digits string.punctuation
5 string.ascii_lowercase string.hexdigits string.whitespace
6 >>> for char in string.digits:
7 ... print(char)
8 ...
9 0
10 1
11 2
12 3
13 4
14 5
15 6
16 7
17 8
18 9
19 >>>
```

The last exercise which we want to leave you with, is to check if something is a consonant.

A consonant is an alphabet which is not a vowel, so any alphabet which is not in `'aeiou'` is a consonant. The simplest way of doing this is to say `consonant_string = 'bcd...'` and so on. Looks like a very long solution? There is an easier way out.

```
1 >>> vowel_string = 'aeiou'
2 >>> char = 'b'
3 >>> char.isalpha() and char.lower() not in vowel_string
4 True
```

## Step 06: More Exercises On Strings

In the step, let's look at a few more puzzles and exercises related to strings. Let's say we have a simple string, `string_example`, and this is contains an English sentence.

```
'This is a great thing.'
```

Let's try to print each of the words present in this string, on a separate line.

So we would want to print `'This'`, `'is'`, `'a'`, `'great'` and `'thing'` on individual lines.

One of the clues we'll give you is, try and do `string_example. <TAB>`. There are a huge list of methods, which would come up if you do that.

```
1 >>> string_example = "This is a great thing"
2 >>> string_example.
3 string_example.capitalize()
4 string_example.casefold()
5 string_example.center()
6 string_example.count()
7 string_example.encode()
8 string_example.endswith()
9 string_example.expandtabs()
10 string_example.find()
11 string_example.format()
12 string_example.format_map()
13 string_example.index()
14 string_example.isalnum()
15 string_example.isalpha()
16 string_example.isdecimal()
17 string_example.isdigit()
18 string_example.isidentifier()
19 string_example.islower()
20 string_example.isnumeric()
21 string_example.isprintable()
22 string_example.isspace()
23 string_example.istitle()
24 string_example.isupper()
```

```
 string_example.join()
 string_example.ljust()
 string_example.lower()
 string_example.lstrip()
 string_example.maketrans()
 string_example.partition()
 string_example.replace()
 string_example.rfind()
 string_example.rindex()
 string_example.rjust()
 string_example.rpartition()
 string_example.rsplit()
 string_example.rstrip()
 string_example.split()
 string_example.splitlines()
 string_example.startswith()
 string_example.strip()
 string_example.swapcase()
 string_example.title()
 string_example.translate()
 string_example.upper()
 string_example.zfill()
```

One of the methods in the list is the `split()` method.

```
1 >>> string_example.split()
2 ['This', 'is', 'a', 'great', 'thing']
```

```
3 >>> for word in string_example.split():
4 ... print(word)
5 ...
6 This
7 is
8 a
9 great
10 thing
```

`split_lines()` method looks for a `'\n'`, and it divides the string based on it. If you have a string which contains newlines, and you would want to divide it into a number of strings with each line as a new element, the method you can use is `split_lines()`.

```
1 >>> string_example = "This\nis\n\ngreat\nnthing"
2 >>> print(string_example)
3 This
4 is
5
6 great
7 thing
8 >>> string_example = "This\nis\nna\ngreat\nnthing"
9 >>> print(string_example)
10 This
11 is
12 a
13 great
14 thing
15 >>> string_example.splitlines()
16 ['This', 'is', 'a', 'great', 'thing']
17 >>>
```

The last thing which we look at, is **concatenation operator**.

```
1 >>> 1 + 2
2 3
3 >>> "1" + "2"
4 '12'
5 >>> "1" + 1
6 TypeError: must be str, not int
7 >>> "ABC" + "DEF"
8 'ABCDEF'
```

In Python, you cannot do `+` operator between two different types. `+` with two strings is concatenation. `+` with two numbers is addition.

One other interesting operator on strings is multiplication. If you do a `'1' * 20`, What do you think will be the output?

```
1 >>> 1 * 20
2 20
3 >>> '1' * 20
4 '11111111111111111111'
5 >>> 'A' * 10
6 'AAAAAAAAAA'
```

If you multiply a string with `number`, the string value is concatenated `number` times.

The last thing which we look at in this step, is comparing strings.

Let's say we have a string with a value `str = 'test'`, and you have another string to with a value `str1 = 'test1'`.

We want to check whether both these strings are the same.

```
1 >>> str = "test"
2 >>> str2 = "test1"
3 >>> str == str2
4 False
5 >>> str2 = "test"
6 >>> str == str2
7 True
```

You can compare strings using the `==` operator.

## Summary

In this step, we explored a few exercises on strings, covering areas such as:

- Splitting a given sentence into individual words

- The concatenation operator, `+`
- The string multiplication pattern, `*`
- The use of the `==` operator to compare strings

## Chapter 07 - Introducing Loops

Welcome to the section on Loops. In this section, we will look at a variety of loops that are available in Python. We will look mainly at the `for` loop, and the `while` loop.

### Step 01: Revisited: The for Loop

Let's start with revising the basics of the for loop, we have learned in the previous steps.

We saw that a `for` loop helps us to loop around the same set of code statements, many times over.

Let's look at a few simple examples, once again.

#### Snippet-01

The syntax of a `for` loop is very simple.

For example, this code snippet will tell you all about it: `for i in range(1, 11): print(i)`.

What does this do? Very simple, it prints from `1` to `10`.

In the call to the `range()` function, the second parameter is exclusive. We are actually looping from `1` to `10`, and this piece of code, `print(i)`, is being executed for different values of `i`.

```
1 >>> for i in range(1,11):
2 ... print(i)
3 ...
4 1
5 2
6 3
7 4
8 5
9 6
10 7
```

```
11 8
12 9
13 10
```

for loop can also be used to loop round the characters in a string.

```
1 >>> for ch in "Hello World":
2 ... print(ch)
3 ...
4 H
5 e
6 l
7 l
8 o
9 W
10 o
11 r
12 l
13 d
```

for loop can be used to loop around all the words in a given sentence.

```
1 >>> for word in "Hello World".split():
2 ... print(word)
3 ...
4 Hello
5 World
```

for loop can be used to loop around a specific list of values.

```
1 >>> for item in (3, 6, 9):
2 ... print(item)
3 ...
4 3
5 6
6 9
```

## Summary

In this step, we started with discussing and revising basic concepts about the `for` loop

### Step 02: Programming Exercise PE-LO-01

Welcome back to this step, where we would do a lot of exercises with the `for` loop.

#### Exercises

1. The first exercise is to find out if a number is prime. We want to write a method, `is_prime()`, which accepts an integer value as parameter, and returns whether it's a prime. (**Hint:** A prime number is something which is only divisible by `1` and itself).
  1. `5` is only divisible by `1` and `5`. It is not divisible by any other number. Same is the case with `7` and `11`.
  2. However, `6` is divisible by `1`, `2`, `3` and `6`. So it's not a prime number.
2. The second exercise is to write a method to calculate the sum up to a given integer, starting from `1`. **Hint:** If I would want to find that the sum up to `6`. what's needed is `1 + 2 + 3 + 4 + 5 + 6`.
3. The third exercise is to find that the sum of divisors of a given integer. **Hint:** Let's say we want to find out the sum of the divisors of `15`. The divisors of `15` are `1`, `3`, `5` and `15`. So I would want to calculate `1 + 3 + 5 + 15`, and return that value.
4. Fourth exercise is to print a numbered triangle, when given a specific integer.

Hint: Given an input `5`, we would want to print the number triangle of these kind:

```
1 1
2 1 2
3 1 2 3
4 1 2 3 4
5 1 2 3 4 5.
```

These are the exercises for the `for` loop. We also test our skills, with creating method and executing them, in our IDE.

#### Solution 1

Let's start with creating the `is_prime()` method, in a file named `for_exercises`.

We would want to accept an `int` parameter, and find out if it is prime, or not.

We need to check whether it's divisible by any other number, other than `1` and itself. If we are passed in a value of `5`, you want to see if it's divisible by any of `2`, `3` or `4`.

```
def is_prime(number):
```

We can use a `for` loop. We can structure it like this:

`for divisor in range(1, number): ...`. We would not want to divide it with `1`, but start with `2` instead, and go up to `number-1`, which is `4`.

```
for divisor in range(2,number):
```

How can we check if the `number` is divisible by `divisor`?

By using the `%` operator. If `number` is divisible by `divisor` we return `False`.

```
1 for divisor in range(2,number):
2 if number % divisor == 0:
3 return False
```

What happens if the code comes up to the end? It would mean we tried with `2`, `3` and `4`, but `number` was not divisible by all of them. In that case, `number` would be prime, and we can safely return `True`.

```
1 for divisor in range(2,number):
2 if number % divisor == 0:
3 return False
4
5 return True
```

For `1`, the rules are a little different, as it is neither a prime or composite. We will add an `if` condition to check if the number is `1`. `if(number < 2):`

This `if` condition is called a guard check or a boundary check, to make sure that you are processing only the right input. If `number` has a value less than `2`, do nothing. OK, it's not a prime.

Here is the entire code at one place, for your reference:

```
1 def is_prime(number):
2 if(number < 2):
3 return False
4 for divisor in range(2,number):
5 if number % divisor == 0:
6 return False
7 return True
8 print(is_prime(5));
```

### Step 03: Continued - Programming Exercise PE-LO-01

In the previous step, we looked at solving the `is_prime()` exercise. In this step, let's look at an implementation of `sum_up_to_n()`. Here is the entire code for this exercise:

```
1 def sum_upto_n(number):
2 sum = 0
3 for i in range(1, number+1):
4 sum = sum + i
5 return sum
6 print(sum_upto_n(6))
7 print(sum_upto_n(10))
```

### Summary

In this step, we:

- Wrote a Python function to compute the sum of all integers, from `1`, up to the input integer `n`.

## Step 04: Continued - Programming Exercise PE-LO-01

Let's focus on the third exercise, `sum_of_divisors` .

One of the clues we can give you, is that `sum_of_divisors()` is very similar to `is_prime()` .

You want to find out if a number is dividing `15` , and if it's dividing `15` , with the remainder of `0` , then you need to add that up.

```
1 def calculate_sum_of_divisors(number):
2 sum = 0
3 if(number < 2):
4 return sum
5 for divisor in range(1,number+1):
6 if number % divisor == 0:
7 sum = sum + divisor
8 return sum
9 print(calculate_sum_of_divisors(6))
10 print(calculate_sum_of_divisors(15))
```

## Step 05: Continued - Programming Exercise PE-LO-01

In this step, Let's look at the last exercise - `print_a_number_triangle` .

For example, if we call such a function with input `5` , the output needs to be:

```
1 1
2
3 1 2
4
5 1 2 3
6
7 1 2 3 4
8
9 1 2 3 4 5
```

Let start with a simple thing. Let's try and print `1 2 3 4 5` first, and then we would look at how to print the rest of the output. Lets proceed with defining this method.

We can say `def print_a_number_triangle(number): ...` that takes a number as an input. You want to print a sequence of integers starting from `1`, up to that specific `number`. How can you do that? Let's try this: `for i in range(1,number+1): print(i)` What would happen? Let's call `print_a_number_triangle(5)` now. It prints:

```
1 1
2 2
3 3
4 4
5 5
```

on individual lines.

To print this sequence on a single line, let's delimit them with `<SPACE>` instead. Call `print()` like this instead: `for i in range(1,number+1): print(i, end=" ")`.

Let's see what would happen now. `1 2 3 4 5`

To solve our exercise, we want to repeat this again and again.

Yes, we need another for loop around it!

```
1 for j in range(1, number+1):
2 for i in range(1, number + 1):
3 print(i, end=" ")
```

Make sure that you have the indentation right. This is called `loop within a loop`.

The output of above program is

```
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Let's add `print("\n")`, so we have a new line at the end of each outer loop iteration.

```
1 for j in range(1, number+1):
```

```
2 for i in range(1, number + 1):
3 print(i, end=" ")
4 print("\n")
```

## Output

```
1 1 2 3 4 5
2 1 2 3 4 5
3 1 2 3 4 5
4 1 2 3 4 5
5 1 2 3 4 5
```

We are printing a square, not a triangle.

What we want to do is to print up to `1` in first line, upto `2` in second line and so on.

How can we do that? Think about it.

When you are inside this loop, you can see the variable `j`.

Instead of `number+1`, let's say `j + 1`.

When `j` has a value of `1`, `for` will print from `1` to `1`. When `j` has a value of `2`, print from `1` to `2`, literally printing `1 2`. When `j` has a value of `3`, I'll print from `1` to `3`. Let's try this and see what would happen.

```
1 for j in range(1, number+1):
2 for i in range(1, j + 1):
3 print(i, end=" ")
4 print("\n")
```

You can see that our number triangle is ready!

```
1 1
2
3 1 2
```

```
4
5 1 2 3
6
7 1 2 3 4
8
9 1 2 3 4 5
```

Here is the entire code for you:

```
1 def print_a_number_triangle(number):
2 for j in range(1, number + 1):
3 for i in range(1, j + 1):
4 print(i, end=' ')
5 print()
6 print_a_number_triangle(6)
```

An important point to note is, a couple of these things can be done in a much simpler way. We will look at these options when we talk about functional programming.

## Summary

In this step, we:

- Presented a solution to the exercise for printing a number triangle.

## Step 06: Introducing The `while` Loop

Let's look at one of the other loops which is present in Python, called the `while` loop.

In the `for` loop, we can specify the range of our iteration, by using the `range()` function.

In a `while` loop, we specify a logical condition. While the condition is true, loop continues running.

Do you remember one place where we use the condition until now? It was in an `if` statement.

Let's see how to use a simple `while` loop.

## Snippet-01:

```
1 >>> i = 5
2 >>> if i == 5:
3 ... print("i is 5")
4 ...
5 i is 5
```

Let's say `i` has a value of `0`, and we then do: `while i < 5: print(i)`.

```
1 >>> i = 0
2 >>> while i < 5:
3 ... print(i)
4 ...
5 0
6 0
7 0
8 0
9 0
10 0
11 0
12 0
13 ^CTraceback (most recent call last):
14 File "<stdin>", line 2, in <module>
15 KeyboardInterrupt
16 >>>
17 KeyboardInterrupt
```

If we leave it to run, you'd see that it continuously prints `0` again, and again. Let's do a `<CTRL-C>` or `<COMMAND-C>` to interrupt this.

What is happening here?

Initially `i` is `0`, and the condition `i < 5` is `True`, and `print(i)` is executed. Next iteration, it checks the condition, it is `True`, and `0` is printed. This continues to happen.

What's happening is an **infinite loop**.

One of the important things to make sure in a `while` loop, is to increment the value of `i`. We need to say something like `i = i + 1`.

```
1 >>> while i < 5:
```

```
2 ... print(i)
3 ... i = i + 1
4 ...
5 0
6 1
7 2
8 3
9 4
```

So how does it work? \* `i` initially had a value of `0`. First the condition is checked. It's `True`, so `0` is printed and then the value of `i` is incremented to `1`.

- `i` is still less than `5`, so the loop continues to execute, and this happens until `4` is printed. `i` again gets incremented to `4 + 1`, or `5`.
- Then we check the condition `i < 5`. This is now `False`. Control goes out of the `while` loop, and terminates it.

When executing a `while`, control flow is just based on a condition. As long as the condition is `True`, we keep executing the code. An important thing to remember, is to make sure the control variable is updated.

```
1 >>> for i in range(0,5): print(i)
2 ...
3 0
4 1
5 2
6 3
7 4
```

A `for` loop is much simpler to code than a `while`. With `while`, we have to write an expression statement, to increment the value.

The question you might have is - What are the situations when you should use a `while`?

We will look at that very soon.

## Summary

In this video, we:

- Were introduced to the concept of a `while` loop in Python
- Understood the importance of a control variable being incremented inside the loop
- Observed differences between the working of a `while`, and a `for` loop

## Step 07: Programming Exercise PE-LO-02

In the previous step, we were introduced to `while` loop. In this step, let's look at a couple of exercises using the `while` loop.

### Exercises

1. `print_squares_upto_limit(30)` : We need to print all the squares of numbers, up to a limit of `30`. The output needs to be `1 4 9 16 25`.
2. `print_cubes_upto_limit(30)` : We need to print all the cubes of numbers, up to a limit of `30`. The output needs to be `1 8 27`.

### Exercise 1: Solution

Here is the entire code for your reference:

```
1 def print_squares_upto_limit(limit):
2 i = 1
3 while i * i < limit:
4 print(i*i, end = " ")
5 i = i + 1
```

Now the next exercise, was to print cubes up to a limit.

The expression in the `while` condition should now be `i*i*i < 30`.

```
1 def print_cubes_upto_limit(limit):
2 i = 1
3 while i * i * i < limit:
4 print(i*i*i, end = " ")
5 i = i + 1
6 print_cubes_upto_limit(80)
```

Could we have implemented above two examples with `for` loop? It would've been a little more difficult.

Typically, we use a `for` loop when we know how many times the loop will be executed is clear at the start.

If we do not know, how many times a loop will run, `while` is a better option.

### Step 08: While Example

Earlier we used `if` statement to implement a solution for this:

- Ask the User for input:
  - Enter two numbers
  - Choose the Option:
    - 1 - Add
    - 2 - Subtract
    - 3 - Multiply
    - 4 - Divide
- Perform the Operation
- Publish the Result

We would want to enhance it to execute in a loop multiple times, until the user chooses to exit. We will add an option 5 - Exit.

- Ask the User for input:
  - Enter two numbers
  - Choose the Option:
    - 1 - Add
    - 2 - Subtract
    - 3 - Multiply
    - 4 - Divide
    - 5 - Exit
- Perform the Operation
- Publish the Result
- Repeat until Option 5 is chosen.

### Snippet-01 Explained

Here's the earlier code we wrote with if:

```
1 number1 = int(input("Enter Number1: "))
2 number2 = int(input("Enter Number2: "))
3
4 print("\n\n1 - Add")
5 print("2 - Subtract")
6 print("3 - Divide")
7 print("4 - Multiply")
8
9 choice = int(input("Choose Operation: "))
10
11 # print(number1 + number2)
12 # print(choice)
13 if choice==1:
14 result = number1 + number2
15 elif choice==2:
16 result = number1 - number2
17 elif choice==3:
18 result = number1 / number2
19 elif choice==4:
20 result = number1 * number2
21 else:
22 result = "Invalid Choice"
23
24 print(result)
```



trinket: run code anywhere

<https://trinket.io/python/fc8a8b85eb>



React App

<http://number-base-converter-react.vercel.app/>



Python Data Structures Repo & Website

/

# Summaries and links for the most relevant projects in the space of Python installation and packaging

## PyPA Projects¶

### bandersnatch¶

[Issues](#) | [GitHub](#) | [PyPI](#)

`bandersnatch` is a PyPI mirroring client designed to efficiently create a complete mirror of the contents of PyPI. Organizations thus save bandwidth and latency on package downloads (especially in the context of automated tests) and to prevent heavily loading PyPI's Content Delivery Network (CDN).

### build¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

`build` is a **PEP 517** compatible Python package builder. It provides a CLI to build packages, as well as a Python API.

### cibuildwheel¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#) | [Discussions](#) | [Discord #cibuildwheel](#)

`cibuildwheel` is a Python package that builds wheels for all common platforms and Python versions on most CI systems. Also see `multibuild`.

### distlib¶

[Docs](#) | [Issues](#) | [Bitbucket](#) | [PyPI](#)

`distlib` is a library which implements low-level functions that relate to packaging and distribution of Python software. `distlib` implements several relevant PEPs (Python Enhancement Proposal standards) and is useful for developers of third-party packaging tools to make and upload binary and source distributions, achieve interoperability, resolve dependencies, manage package resources, and do other similar functions.

Unlike the stricter packaging project (below), which specifically implements modern Python packaging interoperability standards, `distlib` also attempts to provide reasonable fallback behaviours when asked to handle legacy packages and metadata that predate the modern interoperability standards and fall into the subset of packages that are incompatible with those standards.

## **packaging**¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

Core utilities for Python packaging used by pip and setuptools.

The core utilities in the packaging library handle version handling, specifiers, markers, requirements, tags, and similar attributes and tasks for Python packages. Most Python users rely on this library without needing to explicitly call it; developers of the other Python packaging, distribution, and installation tools listed here often use its functionality to parse, discover, and otherwise handle dependency attributes.

This project specifically focuses on implementing the modern Python packaging interoperability standards defined at PyPA specifications, and will report errors for sufficiently old legacy packages that are incompatible with those standards. In contrast, the `distlib` project is a more permissive library that attempts to provide a plausible reading of ambiguous metadata in cases where packaging will instead report an error.

## **pip**¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

The most popular tool for installing Python packages, and the one included with modern versions of Python.

It provides the essential core features for finding, downloading, and installing packages from PyPI and other Python package indexes, and can be incorporated into a wide range of development workflows via its command-line interface (CLI).

## Pipenv¶

[Docs](#) | [Source](#) | [Issues](#) | [PyPI](#)

Pipenv is a project that aims to bring the best of all packaging worlds to the Python world. It harnesses Pipfile, pip, and virtualenv into one single toolchain. It features very pretty terminal colors.

Pipenv aims to help users manage environments, dependencies, and imported packages on the command line. It also works well on Windows (which other tools often underserve), makes and checks file hashes, to ensure compliance with hash-locked dependency specifiers, and eases uninstallation of packages and dependencies. It is used by Python users and system administrators, but has been less maintained since late 2018.

## Pipfile¶

[Source](#)

`Pipfile` and its sister `Pipfile.lock` are a higher-level application-centric alternative to pip's lower-level `requirements.txt` file.

## pipx¶

[Docs](#) | [GitHub](#) | [PyPI](#)

pipx is a tool to install and run Python command-line applications without causing dependency conflicts with other packages installed on the system.

## Python Packaging User Guide¶

[Docs](#) | [Issues](#) | [GitHub](#)

This guide!

## **readme\_renderer¶**

[GitHub](#) and [docs](#) | [PyPI](#)

`readme_renderer` is a library that package developers use to render their user documentation (README) files into HTML from markup languages such as Markdown or reStructuredText. Developers call it on its own or via `twine`, as part of their release management process, to check that their package descriptions will properly display on PyPI.

## **setuptools¶**

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

`setuptools` (which includes `easy_install`) is a collection of enhancements to the Python `distutils` that allow you to more easily build and distribute Python distributions, especially ones that have dependencies on other packages.

`distribute` was a fork of `setuptools` that was merged back into `setuptools` (in v0.7), thereby making `setuptools` the primary choice for Python packaging.

## **trove-classifiers¶**

[Issues](#) | [GitHub](#) | [PyPI](#)

`trove-classifiers` is the canonical source for classifiers on PyPI, which project maintainers use to systematically describe their projects so that users can better find projects that match their needs on the PyPI.

The `trove-classifiers` package contains a list of valid classifiers and deprecated classifiers (which are paired with the classifiers that replace them). Use this package to validate classifiers used in packages intended for uploading to PyPI. As this list of classifiers is published as code, you can install and import it, giving you a more convenient workflow compared to referring to the list published on PyPI. The issue tracker for the project hosts discussions on proposed classifiers and requests for new classifiers.

## **twine¶**

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

Twine is the primary tool developers use to upload packages to the Python Package Index or other Python package indexes. It is a command-line program that passes program files and metadata to a web API. Developers use it because it's the official PyPI upload tool, it's fast and secure, it's maintained, and it reliably works.

## **virtualenv**¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

virtualenv is a tool which uses the command-line path environment variable to create isolated Python Virtual Environments, much as venv does. virtualenv provides additional functionality, compared to venv, by supporting Python 2.7 and by providing convenient features for configuring, maintaining, duplicating, and troubleshooting the virtual environments. For more information, see the section on [Creating Virtual Environments](#).

## **Warehouse**¶

[Docs](#) | [Issues](#) | [GitHub](#)

The current codebase powering the Python Package Index (PyPI). It is hosted at [pypi.org](#). The default source for pip downloads.

## **wheel**¶

[Docs](#) | [Issues](#) | [GitHub](#) | [PyPI](#)

Primarily, the wheel project offers the `bdist_wheel` setuptools extension for creating wheel distributions. Additionally, it offers its own command line utility for creating and installing wheels.

See also auditwheel, a tool that package developers use to check and fix Python packages they are making in the binary wheel format. It provides functionality to discover dependencies, check metadata for compliance, and repair the wheel and metadata to properly link and include external shared libraries in a package.

---

## **Non-PyPA Projects**¶

## **buildout**

[Docs](#) | [Issues](#) | [PyPI](#) | [GitHub](#)

Buildout is a Python-based build system for creating, assembling and deploying applications from multiple parts, some of which may be non-Python-based. It lets you create a buildout configuration and reproduce the same software later.

## **conda**

[Docs](#)

conda is the package management tool for Anaconda Python installations. Anaconda Python is a distribution from Anaconda, Inc specifically aimed at the scientific community, and in particular on Windows where the installation of binary extensions is often difficult.

Conda is a completely separate tool from pip, virtualenv and wheel, but provides many of their combined features in terms of package management, virtual environment management and deployment of binary extensions.

Conda does not install packages from PyPI and can install only from the official Anaconda repositories, or anaconda.org (a place for user-contributed *conda* packages), or a local (e.g. intranet) package server. However, note that pip can be installed into, and work side-by-side with conda for managing distributions from PyPI. Also, conda skeleton is a tool to make Python packages installable by conda by first fetching them from PyPI and modifying their metadata.

## **devpi**

[Docs](#) | [Issues](#) | [PyPI](#)

devpi features a powerful PyPI-compatible server and PyPI proxy cache with a complementary command line tool to drive packaging, testing and release activities with Python. devpi also provides a browsable and searchable web interface.

## **flit**

[Docs](#) | [Issues](#) | [PyPI](#)

Flit provides a simple way to upload pure Python packages and modules to PyPI. It focuses on making the easy things easy for packaging. Flit can generate a configuration file to quickly set up a simple project, build source distributions and wheels, and upload them to PyPI.

Flit uses `pyproject.toml` to configure a project. Flit does not rely on tools such as `setuptools` to build distributions, or `twine` to upload them to PyPI. Flit requires Python 3, but you can use it to distribute modules for Python 2, so long as they can be imported on Python 3.

## **enscons**¶

[Source](#) | [Issues](#) | [PyPI](#)

Enscons is a Python packaging tool based on SCons. It builds pip-compatible source distributions and wheels without using `distutils` or `setuptools`, including distributions with C extensions. Enscons has a different architecture and philosophy than `distutils`. Rather than adding build features to a Python packaging system, enscons adds Python packaging to a general purpose build system. Enscons helps you to build sdists that can be automatically built by `pip`, and wheels that are independent of enscons.

## **Hashdist**¶

[Docs](#) | [GitHub](#)

Hashdist is a library for building non-root software distributions. Hashdist is trying to be “the Debian of choice for cases where Debian technology doesn’t work”. The best way for Pythonistas to think about Hashdist may be a more powerful hybrid of `virtualenv` and `buildout`. It is aimed at solving the problem of installing scientific software, and making package distribution stateless, cached, and branchable. It is used by some researchers but has been lacking in maintenance since 2016.

## **hatch**¶

[GitHub](#) and [Docs](#) | [PyPI](#)

Hatch is a unified command-line tool meant to conveniently manage dependencies and environment isolation for Python developers. Python package developers use Hatch to configure, version, specify dependencies for, and publish packages to PyPI. Under the hood, it uses `twine` to upload packages to PyPI, and `pip` to download and install packages.

## **multibuild**

[GitHub](#)

Multibuild is a set of CI scripts for building and testing Python wheels for Linux, macOS, and (less flexibly) Windows. Also see `cibuildwheel`.

## **pex**

[Docs](#) | [GitHub](#) | [PyPI](#)

pex is both a library and tool for generating `.pex` (Python EXecutable) files, standalone Python environments in the spirit of `virtualenv`. `.pex` files are just carefully constructed zip files with a `#!/usr/bin/env python` and special `__main__.py`, and are designed to make deployment of Python applications as simple as `cp`.

## **pip-tools**

[GitHub](#) and [Docs](#) | [PyPI](#)

pip-tools is a suite of tools meant for Python system administrators and release managers who particularly want to keep their builds deterministic yet stay up to date with new versions of their dependencies. Users can specify particular release of their dependencies via hash, conveniently make a properly formatted list of requirements from information in other parts of their program, update all dependencies (a feature pip currently does not provide), and create layers of constraints for the program to obey.

## **piwheels**

[Website](#) | [Docs](#) | [GitHub](#)

piwheels is a website, and software underpinning it, that fetches source code distribution packages from PyPI and compiles them into binary wheels that are optimized for installation onto Raspberry Pi computers. Raspberry Pi OS pre-configures pip to use `piwheels.org` as an additional index to PyPI.

## **poetry**

[Docs](#) | [GitHub](#) | [PyPI](#)

poetry is a command-line tool to handle dependency installation and isolation as well as building and packaging of Python packages. It uses `pyproject.toml` and, instead of depending on the resolver functionality within pip, provides its own dependency resolver. It attempts to speed users' experience of installation and dependency resolution by locally caching metadata about dependencies.

## **pypiserver**¶

[Docs](#) | [GitHub](#) | [PyPI](#)

pypiserver is a minimalist application that serves as a private Python package index within organizations, implementing a simple API and browser interface. You can upload private packages using standard upload tools, and users can download and install them with pip, without publishing them publicly. Organizations who use pypiserver usually download packages both from pypiserver and from PyPI.

## **scikit-build**¶

[Docs](#) | [GitHub](#) | [PyPI](#)

Scikit-build is an improved build system generator for CPython C/C++/Fortran/Cython extensions that integrates with `setuptools`, `wheel` and `pip`. It internally uses `cmake` (available on PyPI) to provide better support for additional compilers, build systems, cross compilation, and locating dependencies and their associated build requirements. To speed up and parallelize the build of large projects, the user can install `ninja` (also available on PyPI).

## **shiv**¶

[Docs](#) | [GitHub](#) | [PyPI](#)

shiv is a command line utility for building fully self contained Python zipapps as outlined in [PEP 441](#), but with all their dependencies included. Its primary goal is making distributing Python applications and command line tools fast & easy.

## **Spack**¶

[Docs](#) | [GitHub](#) | [Paper](#) | [Slides](#)

A flexible package manager designed to support multiple versions, configurations, platforms, and compilers. Spack is like Homebrew, but packages are written in Python and parameterized to allow easy swapping of compilers, library versions, build options, etc. Arbitrarily many versions of packages can coexist on the same system. Spack was designed for rapidly building high performance scientific applications on clusters and supercomputers.

Spack is not in PyPI (yet), but it requires no installation and can be used immediately after cloning from GitHub.

## **zest.releaser**¶

[Docs](#) | [GitHub](#) | [PyPI](#)

`zest.releaser` is a Python package release tool providing an abstraction layer on top of `twine`. Python developers use `zest.releaser` to automate incrementing package version numbers, updating changelogs, tagging releases in source control, and uploading new packages to PyPI.

---

## **Standard Library Projects**¶

### **ensurepip**¶

[Docs](#) | [Issues](#)

A package in the Python Standard Library that provides support for bootstrapping pip into an existing Python installation or virtual environment. In most cases, end users won't use this module, but rather it will be used during the build of the Python distribution.

### **distutils**¶

[Docs](#) | [Issues](#)

The original Python packaging system, added to the standard library in Python 2.0.

Due to the challenges of maintaining a packaging system where feature updates are tightly coupled to language runtime updates, direct usage of distutils is now actively discouraged, with setuptools being the preferred replacement. setuptools not only provides features that plain distutils doesn't offer (such as dependency declarations and entry point declarations), it also provides a consistent build interface and feature set across all supported Python versions.

## venv¶

[Docs](#) | [Issues](#)

A package in the Python Standard Library (starting with Python 3.3) for creating Virtual Environments. For more information, see the section on [Creating Virtual Environments](#).

# **summary**

# Youtube

## Youtube



**Learn Python - Full Course for Beginners [Tutorial]**

<https://www.youtube.com/watch?v=rfscVS0vtbw>



**CS46 Hash Tables I w/ Tom Tarpey**

<https://youtu.be/mYu3vNKp8SQ>



**CS46 Number Bases and Character Encoding w/ Tom Tarpey**

<https://youtu.be/7bxLc0qwL2c>

<<<<< HEAD



**CS46 Intro to Python I w/ Tom Tarpey**

<https://www.youtube.com/watch?v=bS8X3x2FtK8>



**CS 46 Intro to Python I Augmentation w/ Tom Tarpey**

[https://www.youtube.com/watch?v=Yg0gZuFt\\_0o](https://www.youtube.com/watch?v=Yg0gZuFt_0o)



**CS46 Intro to Python III w/ Tom Tarpey**

<https://www.youtube.com/watch?v=kByGrAty4Z8>



**CS46 Arrays and Strings Python IV w/ Tom Tarpey**

<https://www.youtube.com/watch?v=BJ8YtWWFUnw>



e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

# Running List Of Notes

## Running List Of Notes

<https://lambda-6.gitbook.io/python/>

This Gitbook As A Website

🔗 Python Notes

<https://ds-unit-5-lambda.netlify.app/#>

🔗 <https://golden-lobe-519.notion.site/PYTHON-cb857bd3fa4b4940928842a94dce856d>

<https://golden-lobe-519.notion.site/PYTHON-cb857bd3fa4b4940928842a94dce856d>

My Notion Notes

🔗 DATA\_STRUC\_PYTHON\_NOTES-2

<https://replit.com/@bgoonz/DATASTRUCTURENOTES-2>

### Keywords:

```
1 ***and del for is raise
2 assert elif from lambda return
3 break else global not try
4 class except exec or while
5 continue exec import pass
6 def finally in print***
```

py-notes.pdf

<https://bryan-guner.gitbook.io/notesarchive/>

### DOCS:

<https://docs.python.org/3/>

## <<<<< HEAD

```
1 import math
2
3 def say_hi(name):
4 """<---- Multi-Line Comments and Docstrings
5 This is where you put your content for help() to inform the user
6 about what your function does and how to use it
7 """
8 print(f"Hello {name}!")
9
10 print(say_hi("Bryan")) # Should get the print inside the function, then None
11 # Boolean Values
12 # Work the same as in JS, except they are title case: True and False
13 a = True
14 b = False
15 # Logical Operators
16 # != not, || = or, && = and
17 print(True and True)
18 print(True and not True)
19 print(True or True)
20 # Truthiness - Everything is True except...
21 # False - None, False, '', [], (), set(), range(0)
22 # Number Values
23 # Integers are numbers without a floating decimal point
24 print(type(3)) # type returns the type of whatever argument you pass in
25 # Floating Point values are numbers with a floating decimal point
26 print(type(3.5))
27 # Type Casting
28 # You can convert between ints and floats (along with other types...)
29 print(float(3)) # If you convert a float to an int, it will truncate the dec-
30 print(int(4.5))
31 print(type(str(3)))
32 # Python does not automatically convert types like JS
33 # print(17.0 + ' heyooo ' + 17) # TypeError
34 # Arithmetic Operators
35 # ** - exponent (comparable to Math.pow(num, pow))
36 # // - integer division
37 # There is no ++ or -- in Python
38 # String Values
39 # We can use single quotes, double quotes, or f' ' for string formats
40 # We can use triple single quotes for multiline strings
41 print(
42 """This here's a story
43 All about how
44 My life got twist
45 Turned upside down
46 """
47)
48 # Three double quotes can also be used, but we typically reserve these for
```

```

49 # multi-line comments and function docstrings (refer to lines 6-9)(Nice :D)
50 # We use len() to get the length of something
51 print(len("Bryan G")) # 7 characters
52 print(len(["hey", "ho", "hey", "hey", "ho"])) # 5 list items
53 print(len({1, 2, 3, 4, 5, 6, 7, 9})) # 8 set items
54 # We can index into strings, list, etc..self.
55 name = "Bryan"
56 for i in range(len(name)):
57 print(name[i]) # B, r, y, a, n
58 # We can index starting from the end as well, with negatives
59 occupation = "Full Stack Software Engineer"
60 print(occupation[-3]) # e
61 # We can also get ranges in the index with the [start:stop:step] syntax
62 print(occupation[0:4:1]) # step and stop are optional, stop is exclusive
63 print(occupation[::-4]) # beginning to end, every 4th letter
64 print(occupation[4:14:2]) # Let's get weird with it!
65 # NOTE: Indexing out of range will give you an IndexError
66 # We can also get the index og things with the .index() method, similar to in
67 print(occupation.index("Stack"))
68 print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
69 # We can count how many times a substring/item appears in something as well
70 print(occupation.count("S"))
71 print(
72 """Now this here's a story all about how
73 My life got twist turned upside down
74 I forget the rest but the the potato
75 smells like the potato""".count(
76 "the"
77)
78)
79 # We concatenate the same as Javascript, but we can also multiply strings
80 print("dog " + "show")
81 print("ha" * 10)
82 # We can use format for a multitude of things, from spaces to decimal places
83 first_name = "Bryan"
84 last_name = "Guner"
85 print("Your name is {} {}".format(first_name, last_name))
86 # Useful String Methods
87 print("Hello".upper()) # HELLO
88 print("Hello".lower()) # hello
89 print("HELLO".islower()) # False
90 print("HELLO".isupper()) # True
91 print("Hello".startswith("he")) # False
92 print("Hello".endswith("lo")) # True
93 print("Hello There".split()) # [Hello, There]
94 print("hello1".isalpha()) # False, must consist only of letters
95 print("hello1".isalnum()) # True, must consist of only letters and numbers
96 print("3215235123".isdecimal()) # True, must be all numbers
97 # True, must consist of only spaces/tabs/newlines
98 print("\n ".isspace())
99 # False, index 0 must be upper case and the rest lower
100 print("Bryan Guner".istitle())
101 print("Michael Lee".istitle()) # True!

```

```
102 # Duck Typing - If it walks like a duck, and talks like a duck, it must be a duck
103 # Assignment - All like JS, but there are no special keywords like let or const
104 a = 3
105 b = a
106 c = "heyoo"
107 b = ["reassignment", "is", "fine", "G!"]
108 # Comparison Operators - Python uses the same equality operators as JS, but note:
109 # < - Less than
110 # > - Greater than
111 # <= - Less than or Equal
112 # >= - Greater than or Equal
113 # == - Equal to
114 # != - Not equal to
115 # is - Refers to exact same memory location
116 # not - !
117 # Precedence - Negative Signs(not) are applied first(part of each number)
118 # - Multiplication and Division(and) happen next
119 # - Addition and Subtraction(or) are the last step
120 # NOTE: Be careful when using not along with ==
121 print(not a == b) # True
122 # print(a == not b) # Syntax Error
123 print(a == (not b)) # This fixes it. Answer: False
124 # Python does short-circuit evaluation
125 # Assignment Operators - Mostly the same as JS except Python has **= and //=
126 # Flow Control Statements - if, while, for
127 # Note: Python smushes 'else if' into 'elif'!
128 if 10 < 1:
129 print("We don't get here")
130 elif 10 < 5:
131 print("Nor here...")
132 else:
133 print("Hey there!")
134 # Looping over a string
135 for c in "abcdefghijklmnopqrstuvwxyz":
136 print(c)
137 # Looping over a range
138 for i in range(5):
139 print(i + 1)
140 # Looping over a list
141 lst = [1, 2, 3, 4]
142 for i in lst:
143 print(i)
144 # Looping over a dictionary
145 spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
146 for v in spam.values():
147 print(v)
148 # Loop over a list of tuples and destructuring the values
149 # Assuming spam.items returns a list of tuples each containing two items (k, v)
150 for k, v in spam.items():
151 print(f"[{k}]: {v}")
152 # While loops as long as the condition is True
153 # - Exit loop early with break
154 # - Exit iteration early with continue
```

```
155 spam = 0
156 while True:
157 print("Sike That's the wrong Numba")
158 spam += 1
159 if spam < 5:
160 continue
161 break
162
163 # Functions - use def keyword to define a function in Python
164
165 def printCopyright():
166 print("Copyright 2021, Bgoonz")
167
168 # Lambdas are one liners! (Should be at least, you can use parenthesis to dis-
169 def avg(num1, num2):
170 return print(num1 + num2)
171
172 avg(1, 2)
173 # Calling it with keyword arguments, order does not matter
174 avg(num2=20, num1=1252)
175 printCopyright()
176 # We can give parameters default arguments like JS
177
178 def greeting(name, saying="Hello"):
179 print(saying, name)
180
181 greeting("Mike") # Hello Mike
182 greeting("Bryan", saying="Hello there...")
183 # A common gotcha is using a mutable object for a default parameter
184 # All invocations of the function reference the same mutable object
185
186 def append_item(item_name, item_list=[]): # Will it obey and give us a new l-
187 item_list.append(item_name)
188 return item_list
189
190 # Uses same item list unless otherwise stated which is counterintuitive
191 print(append_item("notebook"))
192 print(append_item("notebook"))
193 print(append_item("notebook", []))
194 # Errors - Unlike JS, if we pass the incorrect amount of arguments to a funct-
195 # it will throw an error
196 # avg(1) # TypeError
197 # avg(1, 2, 2) # TypeError
198 # ----- DAY 2 -----
199 # Functions - * to get rest of position arguments as tuple
200 # - ** to get rest of keyword arguments as a dictionary
201 # Variable Length positional arguments
202
203 def add(a, b, *args):
204 # args is a tuple of the rest of the arguments
205 total = a + b
206 for n in args:
207 total += n
```

```

208 return total
209
210 print(add(1, 2)) # args is None, returns 3
211 print(add(1, 2, 3, 4, 5, 6)) # args is (3, 4, 5, 6), returns 21
212 # Variable Length Keyword Arguments
213
214 def print_names_and_countries(greeting, **kwargs):
215 # kwargs is a dictionary of the rest of the keyword arguments
216 for k, v in kwargs.items():
217 print(greeting, k, "from", v)
218
219 print_names_and_countries(
220 "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
221)
222 # We can combine all of these together
223
224 def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
225 pass
226
227 # Lists are mutable arrays
228 empty_list = []
229 roomates = ["Beau", "Delynn"]
230 # List built-in function makes a list too
231 specials = list()
232 # We can use 'in' to test if something is in the list, like 'includes' in JS
233 print(1 in [1, 2, 4]) # True
234 print(2 in [1, 3, 5]) # False
235 # Dictionaries - Similar to JS POJO's or Map, containing key value pairs
236 a = {"one": 1, "two": 2, "three": 3}
237 b = dict(one=1, two=2, three=3)
238 # Can use 'in' on dictionaries too (for keys)
239 print("one" in a) # True
240 print(3 in b) # False
241 # Sets - Just like JS, unordered collection of distinct objects
242 bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}
243 # bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
244 school_bag = set(
245 ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"])
246)
247 print(school_bag)
248 print(bedroom)
249 # We can use 'in' on sets as well
250 print(1 in {1, 2, 3}) # True
251 print(4 in {1, 3, 5}) # False
252 # Tuples are immutable lists of items
253 time_blocks = ("AM", "PM")
254 colors = "red", "green", "blue" # Parenthesis not needed but encouraged
255 # The tuple built-in function can be used to convert things to tuples
256 print(tuple("abc"))
257 print(tuple([1, 2, 3]))
258 # 'in' may be used on tuples as well
259 print(1 in (1, 2, 3)) # True
260 print(5 in (1, 4, 3)) # False

```

```

261 # Ranges are immutable lists of numbers, often used with for loops
262 # - start - default: 0, first number in sequence
263 # - stop - required, next number past last number in sequence
264 # - step - default: 1, difference between each number in sequence
265 range1 = range(5) # [0,1,2,3,4]
266 range2 = range(1, 5) # [1,2,3,4]
267 range3 = range(0, 25, 5) # [0,5,10,15,20]
268 range4 = range(0) # []
269 for i in range1:
270 print(i)
271 # Built-in functions:
272 # Filter
273
274 def isOdd(num):
275 return num % 2 == 1
276
277 filtered = filter(isOdd, [1, 2, 3, 4])
278 print(list(filtered))
279 for num in filtered:
280 print(f"first way: {num}")
281 print("--" * 20)
282 [print(f"list comprehension: {i}") for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
284 # Map
285
286 def toUpper(str):
287 return str.upper()
288
289 upperCased = map(toUpper, ["a", "b", "c", "d"])
290 print(list(upperCased))
291 # Sorted
292 sorted_items = sorted(["john", "tom", "sonny", "Mike"])
293 print(list(sorted_items)) # Notice uppercase comes before lowercase
294 # Using a key function to control the sorting and make it case insensitive
295 sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
296 print(sorted_items)
297 # You can also reverse the sort
298 sorted_items = sorted(["john", "tom", "sonny", "Mike"],
299 key=str.lower, reverse=True)
300 print(sorted_items)
301 # Enumerate creates a tuple with an index for what you're enumerating
302 quarters = ["First", "Second", "Third", "Fourth"]
303 print(list(enumerate(quarters)))
304 print(list(enumerate(quarters, start=1)))
305 # Zip takes list and combines them as key value pairs, or really however you want
306 keys = ("Name", "Email")
307 values = ("Buster", "cheetoh@johhnydepp.com")
308 zipped = zip(keys, values)
309 print(list(zipped))
310 # You can zip more than 2
311 x_coords = [0, 1, 2, 3, 4]
312 y_coords = [4, 6, 10, 9, 10]
313 z_coords = [20, 10, 5, 9, 1]

```

```
314 coords = zip(x_coords, y_coords, z_coords)
315 print(list(coords))
316 # Len reports the length of strings along with list and any other object data
317 # doing this to save myself some typing
318
319 def print_len(item):
320 return print(len(item))
321
322 print_len("Mike")
323 print_len([1, 5, 2, 10, 3, 10])
324 print_len({1, 5, 10, 9, 10}) # 4 because there is a duplicate here (10)
325 print_len((1, 4, 10, 9, 20))
326 # Max will return the max number in a given scenario
327 print(max(1, 2, 35, 1012, 1))
328 # Min
329 print(min(1, 5, 2, 10))
330 print(min([1, 4, 7, 10]))
331 # Sum
332 print(sum([1, 2, 4]))
333 # Any
334 print(any([True, False, False]))
335 print(any([False, False, False]))
336 # All
337 print(all([True, True, False]))
338 print(all([True, True, True]))
339 # Dir returns all the attributes of an object including it's methods and dunder
340 user = {"Name": "Bob", "Email": "bob@bob.com"}
341 print(dir(user))
342 # Importing packages and modules
343 # - Module - A Python code in a file or directory
344 # - Package - A module which is a directory containing an __init__.py file
345 # - Submodule - A module which is contained within a package
346 # - Name - An exported function, class, or variable in a module
347 # Unlike JS, modules export ALL names contained within them without any special
348 # Assuming we have the following package with four submodules
349 # math
350 # | __init__.py
351 # | addition.py
352 # | subtraction.py
353 # | multiplication.py
354 # | division.py
355 # If we peek into the addition.py file we see there's an add function
356 # addition.py
357 # We can import 'add' from other places because it's a 'name' and is automatically
358
359 # def add(num1, num2):
360 # return num1 + num2
361
362 # Notice the . syntax because this package can import it's own submodules.
363 # Our __init__.py has the following files
364 # This imports the 'add' function
365 # And now it's also re-exported in here as well
366 # from .addition import add
```

```

367 # These import and re-export the rest of the functions from the submodule
368 # from .subtraction import subtract
369 # from .division import divide
370 # from .multiplication import multiply
371 # So if we have a script.py and want to import add, we could do it many ways
372 # This will load and execute the 'math/__init__.py' file and give
373 # us an object with the exported names in 'math/__init__.py'
374 # print(math.add(1,2))
375 # This imports JUST the add from 'math/__init__.py'
376 # from math import add
377 # print(add(1, 2))
378 # This skips importing from 'math/__init__.py' (although it still runs)
379 # and imports directly from the addition.py file
380 # from math.addition import add
381 # This imports all the functions individually from 'math/__init__.py'
382 # from math import add, subtract, multiply, divide
383 # print(add(1, 2))
384 # print(subtract(2, 1))
385 # This imports 'add' renames it to 'add_some_numbers'
386 # from math import add as add_some_numbers
387 # -----
388 # Classes, Methods, and Properties
389
390 class AngryBird:
391 # Slots optimize property access and memory usage and prevent you
392 # from arbitrarily assigning new properties the instance
393 __slots__ = ["_x", "_y"]
394 # Constructor
395
396 def __init__(self, x=0, y=0):
397 # Doc String
398 """
399 Construct a new AngryBird by setting it's position to (0, 0)
400 """
401 # Instance Variables
402 self._x = x
403 self._y = y
404
405 # Instance Method
406
407 def move_up_by(self, delta):
408 self._y += delta
409
410 # Getter
411
412 @property
413 def x(self):
414 return self._x
415
416 # Setter
417
418 @x.setter
419 def x(self, value):

```

```
420 if value < 0:
421 value = 0
422 self._x = value
423
424 @property
425 def y(self):
426 return self._y
427
428 @y.setter
429 def y(self, value):
430 self._y = value
431
432 # Dunder Repr... called by 'print'
433
434 def __repr__(self):
435 return f"<AngryBird ({self._x}, {self._y})>"
```

```
436
437 # JS to Python Classes cheat table
438 # JS Python
439 # constructor() def __init__(self):
440 # super() super().__init__()
441 # this.property self.property
442 # this.method self.method()
443 # method(arg1, arg2){} def method(self, arg1, ...)
444 # get someProperty(){} @property
445 # set someProperty(){} @someProperty.setter
446 # List Comprehensions are a way to transform a list from one format to another
447 # - Pythonic Alternative to using map or filter
448 # - Syntax of a list comprehension
449 # - new_list = [value loop condition]
450 # Using a for loop
451 squares = []
452 for i in range(10):
453 squares.append(i ** 2)
454 print(squares)
455 # value = i ** 2
456 # loop = for i in range(10)
457 squares = [i ** 2 for i in range(10)]
458 print(list(squares))
459 sentence = "the rocket came back from mars"
460 vowels = [character for character in sentence if character in "aeiou"]
461 print(vowels)
462 # You can also use them on dictionaries. We can use the items() method
463 # for the dictionary to loop through it getting the keys and values out at once
464 person = {"name": "Corina", "age": 32, "height": 1.4}
465 # This loops through and capitalizes the first letter of all keys
466 newPerson = {key.title(): value for key, value in person.items()}
467 print(list(newPerson.items()))
```

e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf



## 2.1.7 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

**Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l): # Compute the list of all permutations of l if len(l) <= 1: return [l]
r = []
for i in range(len(l)):
 s = l[:i] + l[i+1:]
 p = perm(s)
 for x in p:
 r.append(l[i:i+1] + x)
return r
```

The following example shows various indentation errors:

```

1 `def perm(l):
2 for i in range(len(l)):
3 s = l[:i] + l[i+1:]
4 p = perm(l[:i] + l[i+1:]) # error: unexpected indent
5 for x in p:
6 r.append(l[i:i+1] + x)
7 return r # error: inconsistent dedent`
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer – the indentation of `return r` does not match a level popped off the stack.)

<https://ds-unit-5-lambda.netlify.app/>

## Python Study Guide for a JavaScript Programmer

Bryan Guner

Mar 5 · 15 min read

Main data types	List operations	List methods
<pre>boolean = True / False integer = 10 float = 10.01 string = "123abc" list = [ value1, value2, ... ] dictionary = { key1:value1, key2:value2, ... }</pre>	<pre>list = [] defines an empty list list[i] = x stores x with index i list[i] retrieves the item with index i list[-1] retrieves last item list[i:j] retrieves items in the range i to j del list[i] removes the item with index i</pre>	<pre>list.append(x) adds x to the end of the list list.extend(L) appends L to the end of the list list.insert(i,x) inserts x at i position list.remove(x) removes the first list item whose value is x list.pop(i) removes the item at position i and returns its value list.clear() removes all items from the list list.index(x) returns a list of values delimited by x list.count(x) returns a string with list values joined by S list.sort() sorts list items list.reverse() reverses list elements list.copy() returns a copy of the list</pre>
Numeric operators	Dictionary operations	String methods
<pre>+ addition - subtraction * multiplication / division ** exponent % modulus // floor division</pre>	<pre>dict = {} defines an empty dictionary dict[k] = x stores x associated to key k dict[k] retrieves the item with key k del dict[k] removes the item with key k</pre>	<pre>string.upper() converts to uppercase string.lower() converts to lowercase string.count(x) counts how many times x appears string.find(x) position of the x first occurrence string.replace(x,y) replaces x for y string.strip(x) returns a list of values delimited by x string.join(L) returns a string with L values joined by string string.format(x) returns a string that includes formatted x</pre>
Boolean operators	Special characters	Dictionary methods
<pre>and logical AND or logical OR not logical NOT</pre>	<pre># coment \n new line \&lt;char&gt; escape char</pre>	<pre>dict.keys() returns a list of keys dict.values() returns a list of values dict.items() returns a list of pairs (key,value) dict.get(k) returns the value associated to the key k dict.pop() removes the item associated to the key and returns its value dict.update(D) adds keys-values (D) to dictionary dict.clear() removes all keys-values from the dictionary dict.copy() returns a copy of the dictionary</pre>
String operations		
<pre>string[i] retrieves character at position i string[-1] retrieves last character string[i:j] retrieves characters in range i to j</pre>		

Legend: x,y stand for any kind of data values, s for a string, n for a number, L for a list where i,j are list indexes, D stands for a dictionary and k is a dictionary key.

[https://miro.medium.com/max/1400/1\\*3V9VOfPk\\_hrFdbEAd3j-QQ.png](https://miro.medium.com/max/1400/1*3V9VOfPk_hrFdbEAd3j-QQ.png)

## Applications of Tutorial & Cheat Sheet Respectively (At Bottom Of Tutorial):

### Basics

- **PEP8** : Python Enhancement Proposals, style-guide for Python.
- `print` is the equivalent of `console.log` .

```
'print() == console.log()'
```

**# is used to make comments in your code.**

```
1 def foo():
2 """
3 The foo function does many amazing things that you
4 should not question. Just accept that it exists and
5 use it with caution.
6 """
7 secretThing()
```

Python has a built in help function that let's you see a description of the source code without having to navigate to it... “-SickNasty ... Autor Unknown”

### Numbers

- Python has three types of numbers:
- **Integer**
- **Positive and Negative Counting Numbers.**

No Decimal Point

Created by a literal non-decimal point number ... or ... with the `int()` constructor.

```
1 print(3) # => 3
2 print(int(19)) # => 19
3 print(int()) # => 0
```

---

### 3. Complex Numbers

Consist of a real part and imaginary part.

#### Boolean is a subtype of integer in Python.

If you came from a background in JavaScript and learned to accept the premise(s) of the following meme...

Than I am sure you will find the means to suspend your disbelief.

```
1 print(2.24) # => 2.24
2 print(2.) # => 2.0
3 print(float()) # => 0.0
4 print(27e-5) # => 0.00027
```

#### KEEP IN MIND:

The i is switched to a j in programming.

*This is because the letter i is common place as the de facto index for any and all enumerable entities so it just makes sense not to compete for name-space when there's another 25 letters that don't get used for every loop under the sun. My most medium apologies to Leonhard Euler.*

```
1 print(7j) # => 7j
2 print(5.1+7.7j)) # => 5.1+7.7j
3 print(complex(3, 5)) # => 3+5j
4 print(complex(17)) # => 17+0j
5 print(complex()) # => 0j
```

- **Type Casting** : The process of converting one number to another.

```
1 # Using Float
2 print(17) # => 17
3 print(float(17)) # => 17.0# Using Int
```

```
4 print(17.0) # => 17.0
5 print(int(17.0)) # => 17# Using Str
6 print(str(17.0) + ' and ' + str(17)) # => 17.0 and 17
```

The arithmetic operators are the same between JS and Python, with two additions:

- `**`: Double asterisk for exponent.
- `//`: Integer Division.
- There are no spaces between math operations in Python.
- Integer Division gives the other part of the number from Module; it is a way to do round down numbers replacing `Math.floor()` in JS.
- There are no `++` and `--` in Python, the only shorthand operators are:

## Strings

- Python uses both single and double quotes.
- You can escape strings like so `'Jodi asked, "What\\\'s up, Sam?"'`
- Multiline strings use triple quotes.

```
1 print('''My instructions are very long so to make them
2 more readable in the code I am putting them on
3 more than one line. I can even include "quotes"
4 of any kind because they won't get confused with
5 the end of the string!'''')
```

Use the `len()` function to get the length of a string.

```
print(len("Spaghetti")) # => 9
```

Python uses zero-based indexing

Python allows negative indexing (thank god!)

```
print("Spaghetti)[-1] # => i print("Spaghetti)[-4] # => e
```

- Python lets you use ranges

You can think of this as roughly equivalent to the slice method called on a JavaScript object or string... (*mind you that in JS ... strings are wrapped in an object (under the hood)... upon which the string methods are actually called. As a immutable primitive type by textbook definition, a string literal could not hope to invoke most of its methods without violating the state it was bound to on initialization if it were not for this bit of syntactic sugar.*)

```
1 print("Spaghetti"[1:4]) # => pag
2 print("Spaghetti"[4:-1]) # => hett
3 print("Spaghetti"[4:4]) # => (empty string)
```

- The end range is exclusive just like `slice` in JS.

```
1 # Shortcut to get from the beginning of a string to a certain index.
2 print("Spaghetti)[:4]) # => Spag
3 print("Spaghetti"][:-1]) # => Spaghett# Shortcut to get from a certain index
4 print("Spaghetti")[1:]) # => paghetti
5 print("Spaghetti"][-4:]) # => etti
```

- The `index` string function is the equiv. of `indexof()` in JS

```
1 print("Spaghetti".index("h")) # => 4
2 print("Spaghetti".index("t")) # => 6
```

- The `count` function finds out how many times a substring appears in a string... pretty nifty for a hard coded feature of the language.

```
1 print("Spaghetti".count("h")) # => 1
2 print("Spaghetti".count("t")) # => 2
3 print("Spaghetti".count("s")) # => 0
4 print('''We choose to go to the moon in this decade and do the other things,
```

```

5 not because they are easy, but because they are hard, because that goal will
6 serve to organize and measure the best of our energies and skills, because tha
7 challenge is one that we are willing to accept, one we are unwilling to
8 postpone, and one which we intend to win, and the others, too.
9 ''' .count('the ') # => 4

```

- You can use `+` to concatenate strings, just like in JS.
- You can also use `*` to repeat strings or multiply strings.
- Use the `format()` function to use placeholders in a string to input values later on.

```

1 first_name = "Billy"
2 last_name = "Bob"
3 print('Your name is {0} {1}'.format(first_name, last_name)) # => Your name is

```

- Shorthand way to use `format` function is:

```
print(f'Your name is {first_name} {last_name}')
```

## Some useful string methods.

- Note that in JS `join` is used on an Array, in Python it is used on String.

Value	Method	Result
<code>s = "Hello"</code>	<code>s.upper()</code>	<code>"HELLO"</code>
<code>s = "Hello"</code>	<code>s.lower()</code>	<code>"hello"</code>
<code>s = "Hello"</code>	<code>s.islower()</code>	<code>False</code>
<code>s = "hello"</code>	<code>s.islower()</code>	<code>True</code>
<code>s = "Hello"</code>	<code>s.isupper()</code>	<code>False</code>
<code>s = "HELLO"</code>	<code>s.isupper()</code>	<code>True</code>
<code>s = "Hello"</code>	<code>s.startswith("He")</code>	<code>True</code>
<code>s = "Hello"</code>	<code>s.endswith("lo")</code>	<code>True</code>
<code>s = "Hello World"</code>	<code>s.split()</code>	<code>[ "Hello", "World" ]</code>
<code>s = "i-am-a-dog"</code>	<code>s.split("-")</code>	<code>[ "i", "am", "a", "dog" ]</code>

[https://miro.medium.com/max/630/0\\*eE3E5H0AoqkhqK1z.png](https://miro.medium.com/max/630/0*eE3E5H0AoqkhqK1z.png)

- There are also many handy testing methods.

Method	Purpose
<code>isalpha()</code>	returns <code>True</code> if the string consists only of letters and is not blank.
<code>isalnum()</code>	returns <code>True</code> if the string consists only of letters and numbers and is not blank.
<code>isdecimal()</code>	returns <code>True</code> if the string consists only of numeric characters and is not blank.
<code>isspace()</code>	returns <code>True</code> if the string consists only of spaces, tabs, and newlines and is not blank.
<code>istitle()</code>	returns <code>True</code> if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

[https://miro.medium.com/max/630/0\\*Q0CMqFd4PozLDFPB.png](https://miro.medium.com/max/630/0*Q0CMqFd4PozLDFPB.png)

## Variables and Expressions

- **Duck-Typing** : Programming Style which avoids checking an object's type to figure out what it can do.
- Duck Typing is the fundamental approach of Python.
- Assignment of a value automatically declares a variable.

```

1 a = 7
2 b = 'Marbles'
3 print(a) # => 7
4 print(b) # => Marbles

```

- *You can chain variable assignments to give multiple var names the same value.*

Use with caution as this is highly unreadable

```

1 count = max = min = 0
2 print(count) # => 0
3 print(max) # => 0
4 print(min) # => 0

```

The value and type of a variable can be re-assigned at any time.

```

1 a = 17
2 print(a) # => 17

```

```
3 a = 'seventeen'
4 print(a) # => seventeen
```

- `NaN` does not exist in Python, but you can 'create' it like so: `print(float("nan"))`
- Python replaces `null` with `None`.
- `None` is an object and can be directly assigned to a variable.

Using `None` is a convenient way to check to see why an action may not be operating correctly in your program.

## Boolean Data Type

- One of the biggest benefits of Python is that it reads more like English than JS does.

Python	JavaScript
<code>and</code>	<code>&amp;&amp;</code>
<code>or</code>	<code>  </code>
<code>not</code>	<code>!</code>

[https://miro.medium.com/max/1400/0\\*HQpndNhm1Z\\_xSoHb.png](https://miro.medium.com/max/1400/0*HQpndNhm1Z_xSoHb.png)

```
1 # Logical AND
2 print(True and True) # => True
3 print(True and False) # => False
4 print(False and False) # => False# Logical OR
5 print(True or True) # => True
6 print(True or False) # => True
7 print(False or False) # => False# Logical NOT
8 print(not True) # => False
9 print(not False and True) # => True
10 print(not True or False) # => False
```

- By default, Python considers an object to be true UNLESS it is one of the following:
- Constant `None` or `False`
- Zero of any numeric type.
- Empty Sequence or Collection.

- `True` and `False` must be capitalized

## Comparison Operators

- Python uses all the same equality operators as JS.
- In Python, equality operators are processed from left to right.
- Logical operators are processed in this order:
  - **NOT**
  - **AND**
  - **OR**

Just like in JS, you can use parentheses to change the inherent order of operations. Short Circuit : Stopping a program when a true or false has been reached.

Expression	Right side evaluated?
<code>True</code> and ...	Yes
<code>False</code> and ...	No
<code>True</code> or ...	No
<code>False</code> or ...	Yes

[https://miro.medium.com/max/630/0\\*qHzGRLTOMTf30miT.png](https://miro.medium.com/max/630/0*qHzGRLTOMTf30miT.png)

## Identity vs Equality

\*\*\*\*

```

1 print (2 == '2') # => False
2 print (2 is '2') # => Falseprint ("2" == '2') # => True
3 print ("2" is '2') # => True# There is a distinction between the number type
4 print (2 == 2.0) # => True
5 print (2 is 2.0) # => False

```

- In the Python community it is better to use `is` and `is not` over `==` or `!=`

## If Statements

```
if name == 'Monica': print('Hi, Monica.')
if name == 'Monica': print('Hi, Monica.')
else: print('Hello, stranger.')
if name == 'Monica': print('Hi, Monica.')
elif age < 12: print('You are not Monica, kiddo.')
elif age > 2000: print('Unlike you, Monica is not an undead, immortal vampire.')
elif age > 100: print('You are not Monica, grannie.')
Remember the order of elif statements matter.
```

## While Statements

```
1 spam = 0
2 while spam < 5:
3 print('Hello, world.')
4 spam = spam + 1
```

- Break statement also exists in Python.

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam >= 5:
6 break
```

- As are continue statements

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam < 5:
6 continue
7 break
```

## Try/Except Statements

- Python equivalent to `try/catch`

```
1 a = 321
2 try:
3 print(len(a))
4 except:
5 print('Silently handle error here') # Optionally include a correction
6 a = str(a)
7 print(len(a)a = '321'
8 try:
9 print(len(a))
10 except:
11 print('Silently handle error here') # Optionally include a correction
12 a = str(a)
13 print(len(a))
```

- You can name an error to give the output more specificity.

```
1 a = 100
2 b = 0
3 try:
4 c = a / b
5 except ZeroDivisionError:
6 c = None
7 print(c)
```

- You can also use the `pass` command to bypass a certain error.

```
1 a = 100
2 b = 0
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
```

- The `pass` method won't allow you to bypass every single error so you can chain an exception series like so:

```
1 a = 100
2 # b = "5"
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
7 except (TypeError, NameError):
8 print("ERROR!")
```

- You can use an `else` statement to end a chain of `except` statements.

```
1 # tuple of file names
2 files = ('one.txt', 'two.txt', 'three.txt')# simple loop
3 for filename in files:
4 try:
5 # open the file in read mode
6 f = open(filename, 'r')
7 except OSError:
8 # handle the case where file does not exist or permission is denied
9 print('cannot open file', filename)
10 else:
11 # do stuff with the file object (f)
12 print(filename, 'opened successfully')
13 print('found', len(f.readlines()), 'lines')
14 f.close()
```

- `finally` is used at the end to clean up all actions under any circumstance.

```
1 def divide(x, y):
2 try:
3 result = x / y
4 except ZeroDivisionError:
5 print("Cannot divide by zero")
6 else:
7 print("Result is", result)
8 finally:
9 print("Finally...")
```

- Using duck typing to check to see if some value is able to use a certain method.

```
1 # Try a number - nothing will print out
2 a = 321
3 if hasattr(a, '__len__'):
4 print(len(a))# Try a string - the length will print out (4 in this case)
5 b = "5555"
6 if hasattr(b, '__len__'):
7 print(len(b))
```

## Pass

- Pass Keyword is required to write the JS equivalent of :

```
1 if (true) {
2 }while (true) {}if True:
3 passwhile True:
4 pass
```

## Functions

- Function definition includes:
- The `def` keyword
- The name of the function
- A list of parameters enclosed in parentheses.
- A colon at the end of the line.
- One tab indentation for the code to run.
- You can use default parameters just like in JS

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)greeting("Monica")
3 # Hello Monica
4 greeting("Barry", "Hey")
Hey Barry
```

**Keep in mind, default parameters must always come after regular parameters.**

```
1 # THIS IS BAD CODE AND WILL NOT RUN
2 def increment(delta=1, value):
```

```
3 return delta + value
```

- You can specify arguments by name without destructuring in Python.

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)# name has no default value, so just provide the value
3 # saying has a default value, so use a keyword argument
4 greeting("Monica", saying="Hi")
```

- The `lambda` keyword is used to create anonymous functions and are supposed to be one-liners .

```
toUpper = lambda s: s.upper()
```

## Notes

### Formatted Strings

Remember that in Python `join()` is called on a string with an array/list passed in as the argument. Python has a very powerful formatting engine. `format()` is also applied directly to strings.

```
1 shopping_list = ['bread', 'milk', 'eggs']
2 print(', '.join(shopping_list))
```

### Comma Thousands Separator

```
1 print('{:,}'.format(1234567890))
2 '1,234,567,890'
```

### Date and Time

```
1 d = datetime.datetime(2020, 7, 4, 12, 15, 58)
2 print('{:%Y-%m-%d %H:%M:%S}'.format(d))
3 '2020-07-04 12:15:58'
```

## Percentage

```
1 points = 190
2 total = 220
3 print('Correct answers: {:.2%}'.format(points/total))
4 Correct answers: 86.36%
```

## Data Tables

```
1 width=8
2 print(' decimal hex binary')
3 print('-'*27)
4 for num in range(1,16):
5 for base in 'dXb':
6 print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
7 print()
8 Getting Input from the Command Line
9 Python runs synchronously, all programs and processes will stop when listening
10 The input function shows a prompt to a user and waits for them to type 'ENTER'
11 Scripts vs Programs
12 Programming Script : A set of code that runs in a linear fashion.
13 The largest difference between scripts and programs is the level of complexity
```

\*\*Python can be used to display html, css, and JS.\*\**It is common to use Python as an API (Application Programming Interface)*

## Structured Data

**Sequence :** The most basic data structure in Python where the index determines the order.

| List-Tuple-Range-Collections : Unordered data structures, hashable values.

**Dictionaries-Sets-Iterable :** Generic name for a sequence or collection; any object that can be iterated through. Can be mutable or immutable. Built In Data Types

**Lists are the python equivalent of arrays.**

```
1 empty_list = []
2 departments = ['HR', 'Development', 'Sales', 'Finance', 'IT', 'Customer Support']
```

**You can instantiate**

```
specials = list()
```

**Test if a value is in a list.**

```
1 print(1 in [1, 2, 3]) #> True
2 print(4 in [1, 2, 3]) #> False
3 # Tuples : Very similar to lists, but they are immutable
```

**Instantiated with parentheses**

```
time_blocks = ('AM', 'PM')
```

**Sometimes instantiated without**

```
1 colors = 'red','blue','green'
2 numbers = 1, 2, 3
```

**Tuple() built in can be used to convert other data into a tuple**

```
1 tuple('abc') # returns ('a', 'b', 'c')
2 tuple([1,2,3]) # returns (1, 2, 3)
3 # Think of tuples as constant variables.
```

**Ranges : A list of numbers which can't be changed; often used with for loops.**

**Declared using one to three parameters.**

Start : opt. default 0, first # in sequence. Stop : required next number past the last number in the sequence. Step : opt. default 1, difference between each number in the sequence.

```
1 range(5) # [0, 1, 2, 3, 4]
2 range(1,5) # [1, 2, 3, 4]
3 range(0, 25, 5) # [0, 5, 10, 15, 20]
4 range(0) # []
5 for let (i = 0; i < 5; i++)
6 for let (i = 1; i < 5; i++)
7 for let (i = 0; i < 25; i+=5)
8 for let(i = 0; i = 0; i++)
9 # Keep in mind that stop is not included in the range.
```

**Dictionaries : Mappable collection where a hashable value is used as a key to ref. an object stored in the dictionary.**

**Mutable.**

```
1 a = {'one':1, 'two':2, 'three':3}
2 b = dict(one=1, two=2, three=3)
3 c = dict([('two', 2), ('one', 1), ('three', 3)])
4 # a, b, and c are all equal
```

***Declared with curly braces of the built in dict()***

Benefit of dictionaries in Python is that it doesn't matter how it is defined, if the keys and values are the same the dictionaries are considered equal.

**Use the in operator to see if a key exists in a dictionary.**

**Sets : Unordered collection of distinct objects; objects that need to be hashable.**

Always be unique, duplicate items are auto dropped from the set.

## Common Uses:

Removing DuplicatesMembership TestingMathematical Operators: Intersection, Union, Difference, Symmetric Difference.

**Standard Set is mutable, Python has a immutable version called frozenset. Sets created by putting comma seperated values inside braces:**

```
1 school_bag = {'book', 'paper', 'pencil', 'pencil', 'book', 'book', 'book', 'eraser'}
2 print(school_bag)
```

Also can use set constructor to automatically put it into a set.

```
1 letters = set('abracadabra')
2 print(letters)
3 #Built-In Functions
4 #Functions using iterables
```

**filter(function, iterable) : creates new iterable of the same type which includes each item for which the function returns true.**

**map(function, iterable) : creates new iterable of the same type which includes the result of calling the function on every item of the iterable.**

**sorted(iterable, key=None, reverse=False) : creates a new sorted list from the items in the iterable.**

**Output is always a list**

**key: opt function which converts and item to a value to be compared.**

**reverse: optional boolean.**

**enumerate(iterable, start=0) : starts with a sequence and converts it to a series of tuples**

```
1 quarters = ['First', 'Second', 'Third', 'Fourth']
2 print(enumerate(quarters))
3 print(enumerate(quarters, start=1))
```

(0, 'First'), (1, 'Second'), (2, 'Third'), (3, 'Fourth')

(1, 'First'), (2, 'Second'), (3, 'Third'), (4, 'Fourth')

`zip(*iterables)` : creates a zip object filled with tuples that combine 1 to 1 the items in each provided iterable.

Functions that analyze iterable

**len(iterable)** : returns the count of the number of items.

*\*`max(args, key=None)` : returns the largest of two or more arguments.*

**max(iterable, key=None)** : returns the largest item in the iterable.

*key optional function which converts an item to a value to be compared.* `min` works the same way as `max`

**sum(iterable)** : used with a list of numbers to generate the total.

*There is a faster way to concatenate an array of strings into one string, so do not use sum for that.*

**any(iterable)** : returns True if any items in the iterable are true.

**all(iterable)** : returns True is all items in the iterable are true.

## Working with dictionaries

**dir(dictionary)** : returns the list of keys in the dictionary.

**Working with sets**

*\*Union : The pipe / operator or `union(sets)` function can be used to produce a new set which is a combination of all elements in the provided set.*

```
1 a = {1, 2, 3}
```

```
2 b = {2, 4, 6}
3 print(a | b) # => {1, 2, 3, 4, 6}
```

**Intersection :** The & operator can be used to produce a new set of only the elements that appear in all sets.

```
1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a & b) # => {2}
4 Difference : The - operator can be used to produce a new set of only the elements
```

**Symmetric Difference :** The ^ operator can be used to produce a new set of only the elements that appear in exactly one set and not in both.

```
1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a - b) # => {1, 3}
4 print(b - a) # => {4, 6}
5 print(a ^ b) # => {1, 3, 4, 6}
```

**For Statements** In python, there is only one for loop.

Always Includes:

1. The for keyword
2. A variable name
3. The 'in' keyword
4. An iterable of some kind
5. A colon
6. On the next line, an indented block of code called the for clause.

You can use break and continue statements inside for loops as well.

You can use the range function as the iterable for the for loop.

```
1 print('My name is')
2 for i in range(5):
3 print('Carlita Cinco (' + str(i) + ')')
4 total = 0
5 for num in range(101):
6 total += num
```

```
6 print(total)
7 Looping over a list in Python
8 for c in ['a', 'b', 'c']:
9 print(c)lst = [0, 1, 2, 3]
10 for i in lst:
11 print(i)
```

**Common technique is to use the `len()` on a pre-defined list with a `for` loop to iterate over the indices of the list.**

```
1 supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
2 for i in range(len(supplies)):
3 print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

\*\*\*\*

**You can loop and destructure at the same time.**

```
1 l = 1, 2], [3, 4], [5, 6
2 for a, b in l:
3 print(a, ', ', b)
```

Prints 1, 2Prints 3, 4Prints 5, 6

**You can use `values()` and `keys()` to loop over dictionaries.**

```
1 spam = {'color': 'red', 'age': 42}
2 for v in spam.values():
3 print(v)
```

*Prints red*

*Prints 42*

```
1 for k in spam.keys():
2 print(k)
```

*Prints color*

*Prints age*

**For loops can also iterate over both keys and values.**

**Getting tuples**

```
1 for i in spam.items():
2 print(i)
```

*Prints ('color', 'red')*

*Prints ('age', 42)*

**Destructuring to values**

```
1 for k, v in spam.items():
2 print('Key: ' + k + ' Value: ' + str(v))
```

*Prints Key: age Value: 42*

*Prints Key: color Value: red*

**Looping over string**

```
1 for c in "abcdefg":
2 print(c)
```

**When you order arguments within a function or function call, the args need to occur in a particular order:**

*formal positional args.*

- args

*keyword args with default values*

- \*kwargs

```
1 def example(arg_1, arg_2, *args, **kwargs):
2 pass
def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blowfish", **kwargs):
 pass
```

## Importing in Python

**Modules are similar to packages in Node.js** Come in different types:

Built-In,

Third-Party,

Custom.

**All loaded using import statements.**

## Terms

module : Python code in a separate file.package : Path to a directory that contains modules.init.py : Default file for a package.submodule : Another file in a module's folder.function : Function in a module.

**A module can be any file but it is usually created by placing a special file init.py into a folder.**  
**pic**

*Try to avoid importing with wildcards in Python.*

*Use multiple lines for clarity when importing.*

```
1 from urllib.request import (
2 HTTPDefaultErrorHandler as ErrorHandler,
3 HTTPRedirectHandler as RedirectHandler,
4 Request,
5 pathname2url,
6 url2pathname,
7 urlopen,
8)
```

## Watching Out for Python 2

**Python 3 removed <> and only uses !=**

**format() was introduced with P3**

**All strings in P3 are unicode and encoded.md5 was removed.**

**ConfigParser was renamed to configparsersets were killed in favor of set() class.**

**print was a statement in P2, but is a function in P3.**

<https://gist.github.com/bgoonz/82154f50603f73826c27377ebaa498b5#file-python-study-guide-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

<https://gist.github.com/bgoonz/999163a278b987fe47fb247fd4d66904#file-python-cheat-sheet-md>



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/be5715e2-c834-458f-8c5b-ea185717fe37/Untitled.png>

## Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

## Builtin Methods

**abs** (*x*)Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing

`[__abs__()](<https://docs.python.org/3/reference/datamodel.html#object.__abs__>)`.

If the argument is a complex number, its magnitude is returned.

**all** (*iterable*)Return `True` if all elements of the *iterable* are true (or if the iterable is empty).

Equivalent to:

\*\*

```
def* all(iterable): **for** element **in** iterable: **if** **not** element:
return **Falsereturn** **True**
```

**any** (*iterable*)Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

\*\*

```
def* any(iterable): **for** element **in** iterable: **if** element: **return**
Truereturn **False**
```

**ascii** (*object*)As `[repr()](<https://docs.python.org/3/library/functions.html#repr>)`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by

`[repr()](<https://docs.python.org/3/library/functions.html#repr>)` using `\x`, `\u` or `\u` escapes. This generates a string similar to that returned by

`[repr()](<https://docs.python.org/3/library/functions.html#repr>)` in Python 2.

**bin** (*x*)Convert an integer number to a binary string prefixed with "0b". The result is a valid Python expression. If *x* is not a Python

`[int](<https://docs.python.org/3/library/functions.html#int>)` object, it has to define an

```
[__index__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

method that returns an integer. Some examples:>>>

```
** >>>** bin(3) '0b11' **>>>** bin(-10) '-0b1010'
```

If prefix "0b" is desired or not, you can use either of the following ways.>>>

```
**
```

```
>>>** format(14, '#b'), format(14, 'b') ('0b1110', '1110') **>>>** f'*{*14*:#b*}*', f'*{*14*:#b*}* ('0b1110', '1110')
```

See also [\[format\(\)\]\(<https://docs.python.org/3/library/functions.html#format>\)](https://docs.python.org/3/library/functions.html#format) for more information.

**class bool ([x])**Return a Boolean value, i.e. one of `True` or `False`. `x` is converted using the standard truth testing procedure. If `x` is false or omitted, this returns `False`; otherwise it returns `True`. The [\[bool\]\(<https://docs.python.org/3/library/functions.html#bool>\)](https://docs.python.org/3/library/functions.html#bool) class is a subclass of [\[int\]\(<https://docs.python.org/3/library/functions.html#int>\)](https://docs.python.org/3/library/functions.html#int) (see Numeric Types – `int`, `float`, `complex`). It cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Values). *Changed in version 3.7:* `x` is now a positional-only parameter.

**breakpoint (\*args, \*\*kws)**This function drops you into the debugger at the call site.

Specifically, it calls

```
[sys.breakpointhook()]
(<https://docs.python.org/3/library/sys.html#sys.breakpointhook>)
```

, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls [\[pdb.set\\_trace\(\)\]\(<https://docs.python.org/3/library/pdb.html#pdb.set\\_trace>\)](https://docs.python.org/3/library/pdb.html#pdb.set_trace) expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import [\[pdb\]\(<https://docs.python.org/3/library/pdb.html#module-pdb>\)](https://docs.python.org/3/library/pdb.html#module-pdb) or type as much code to enter the debugger. However,

```
[sys.breakpointhook()]
(<https://docs.python.org/3/library/sys.html#sys.breakpointhook>)
```

can be set to some other function and

[\[breakpoint\(\)\]\(<https://docs.python.org/3/library/functions.html#breakpoint>\)](https://docs.python.org/3/library/functions.html#breakpoint) will automatically call that, allowing you to drop into the debugger of choice. Raises an auditing event `builtins.breakpoint` with argument `breakpointhook`. *New in version 3.7.*

**class bytearray ([source[, encoding[, errors]]])**Return a new array of bytes. The [\[bytearray\]\(<https://docs.python.org/3/library/stdtypes.html#bytearray>\)](https://docs.python.org/3/library/stdtypes.html#bytearray) class is a mutable sequence of integers in the range  $0 \leq x < 256$ . It has most of the usual methods of

mutable sequences, described in [Mutable Sequence Types](#), as well as most methods that the `[bytes](<https://docs.python.org/3/library/stdtypes.html#bytes>)` type has, see [Bytes and Bytearray Operations](#). The optional `source` parameter can be used to initialize the array in a few different ways:

- If it is a `string`, you must also give the `encoding` (and optionally, `errors`) parameters;

`[bytearray()](<https://docs.python.org/3/library/stdtypes.html#bytearray>)` then converts the string to bytes using

`[str.encode()](<https://docs.python.org/3/library/stdtypes.html#str.encode>)` . • If it is an `integer`, the array will have that size and will be initialized with null bytes. • If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array. • If it is an `iterable`, it must be an iterable of integers in the range `0 <= x < 256` , which are used as the initial contents of the array. Without an argument, an array of size 0 is created. See also [Binary Sequence Types – bytes, bytearray, memoryview and Bytearray Objects](#).

`class bytes ([source, encoding, errors])`)Return a new “bytes” object, which is an immutable sequence of integers in the range `0 <= x < 256` .

`[bytes](<https://docs.python.org/3/library/stdtypes.html#bytes>)` is an immutable version of `[bytearray](<https://docs.python.org/3/library/stdtypes.html#bytearray>)` – it has the same non-mutating methods and the same indexing and slicing behavior. Accordingly, constructor arguments are interpreted as for

`[bytearray()](<https://docs.python.org/3/library/stdtypes.html#bytearray>)` . Bytes objects can also be created with literals, see [String and Bytes literals](#). See also [Binary Sequence Types – bytes, bytearray, memoryview, Bytes Objects, and Bytes and Bytearray Operations](#).

`callable (object)`Return

`[True](<https://docs.python.org/3/library/constants.html#True>)` if the `object` argument appears callable,

`[False](<https://docs.python.org/3/library/constants.html#False>)` if not. If this returns `True` , it is still possible that a call fails, but if it is `False` , calling `object` will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a

`[__call__()](<https://docs.python.org/3/reference/datamodel.html#object.__call__>)` method. *New in version 3.2:* This function was first removed in Python 3.0 and then brought back in Python 3.2.

`chr (i)`Return the string representing a character whose Unicode code point is the integer `i`. For example, `chr(97)` returns the string `'a'` , while `chr(8364)` returns the string `'€'` . This is

the inverse of `[ord()]` (<https://docs.python.org/3/library/functions.html#ord>) . The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `[ValueError]` (<https://docs.python.org/3/library/exceptions.html#ValueError>) will be raised if *i* is outside that range.

`@**classmethod**` Transform a method into a class method. A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
** class** **C**: **@classmethoddef** f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function decorator – see Function definitions for details. A class method can be called either on the class (such as `c.f()` ) or on an instance (such as `c().f()` ). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument. Class methods are different than C++ or Java static methods. If you want those, see

`[staticmethod()]` (<https://docs.python.org/3/library/functions.html#staticmethod>) in this section. For more information on class methods, see The standard type hierarchy.

*Changed in version 3.9:* Class methods can now wrap other descriptors such as

```
[property()]
```

`compile (source, filename, mode, flags=0, dont_inherit=False, optimize=-1)` Compile the *source* into a code or AST object. Code objects can be executed by

```
[exec()]
```

`[eval()]` (<https://docs.python.org/3/library/functions.html#eval>) . *source* can either be a normal string, a byte string, or an AST object. Refer to the

```
[ast]
```

module documentation for information on how to work with AST objects. The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used). The *mode* argument specifies what kind of code must be compiled; it can be `'exec'` if *source* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed). The optional arguments *flags* and *dont\_inherit* control which compiler options should be activated and which future features should be allowed. If neither is present (or both are zero) the code is compiled with the same flags that affect the code that is calling `[compile()]` (<https://docs.python.org/3/library/functions.html#compile>) . If the *flags*

argument is given and `dont_inherit` is not (or is zero) then the compiler options and the future statements specified by the `flags` argument are used in addition to those that would be used anyway. If `dont_inherit` is a non-zero integer then the `flags` argument is it – the flags (future features and compiler options) in the surrounding code are ignored. Compiler options and future statements are specified by bits which can be bitwise ORed together to specify multiple options. The bitfield required to specify a given future feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the

```
[__future__](<https://docs.python.org/3/library/__future__.html#module-__future__>)
```

module. Compiler flags can be found in

`[ast]`(<https://docs.python.org/3/library/ast.html#module-ast>) module, with `PyCF_` prefix. The argument `optimize` specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by

```
[-0](<https://docs.python.org/3/using/cmdline.html#cmdoption-o>) options. Explicit levels are 0 (no optimization; __debug__ is true), 1 (asserts are removed, __debug__ is false) or 2 (docstrings are removed too). This function raises
```

```
[SyntaxError](<https://docs.python.org/3/library/exceptions.html#SyntaxError>) if the compiled source is invalid, and
```

```
[ValueError](<https://docs.python.org/3/library/exceptions.html#ValueError>) if the source contains null bytes. If you want to parse Python code into its AST representation, see [ast.parse()](<https://docs.python.org/3/library/ast.html#ast.parse>).
```

Raises an auditing event `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

**Note** When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the

```
[code](<https://docs.python.org/3/library/code.html#module-code>) module. Warning It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler. Changed in version 3.2: Allowed use of Windows and Mac newlines. Also input in 'exec' mode does not have to end in a newline anymore. Added the optimize parameter. Changed in version 3.5: Previously,
```

```
[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>) was raised when null bytes were encountered in source. New in version 3.8:
```

```
ast.PyCF_ALLOW_TOP_LEVEL_AWAIT can now be passed in flags to enable support for top-level await, async for, and async with.
```

`class complex ([real[, imag]])`) Return a complex number with the value `real + *imag**1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If `imag` is omitted, it defaults to zero and the constructor serves as a numeric conversion like

`[int](<https://docs.python.org/3/library/functions.html#int>)` and  
`[float](<https://docs.python.org/3/library/functions.html#float>)`. If both arguments are omitted, returns `0j`. For a general Python object `x`, `complex(x)` delegates to  
`x.__complex__()`. If `__complex__()` is not defined then it falls back to  
`__float__()`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_float\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__float__))  
. If `__float__()` is not defined then it falls back to

`__index__()`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_index\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__index__))

. **Note** When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `[ValueError]`(<https://docs.python.org/3/library/exceptions.html#ValueError>). The complex type is described in Numeric Types – int, float, complex. *Changed in version 3.6:* Grouping digits with underscores as in code literals is allowed. *Changed in version 3.8:* Falls back to

`__index__()`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_index\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__index__))

if

`__complex__()`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_complex\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__complex__))

and

`__float__()`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_float\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__float__))

are not defined.

`delattr (object, name)`This is a relative of

`[setattr()](<https://docs.python.org/3/library/functions.html setattr>)`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x foobar`.

`class dict (**kwargs)``class dict (mapping, **kwargs)``class dict (iterable, **kwargs)`Create a new dictionary. The `[dict](<https://docs.python.org/3/library/stdtypes.html#dict>)` object is the dictionary class. See

`[dict](<https://docs.python.org/3/library/stdtypes.html#dict>)` and Mapping Types – `dict` for documentation about this class. For other containers see the built-in

`[list](<https://docs.python.org/3/library/stdtypes.html#list>)`,

`[set](<https://docs.python.org/3/library/stdtypes.html#set>)`, and

`[tuple](<https://docs.python.org/3/library/stdtypes.html#tuple>)` classes, as well as the

`[collections](<https://docs.python.org/3/library/collections.html#module-collections>)`

module.

`dir ([object])`Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object. If the object has a method named

`[__dir__()](<https://docs.python.org/3/reference/datamodel.html#object.__dir__>)`, this method will be called and must return the list of attributes. This allows objects that implement a custom

`[__getattr__()]`  
`(<https://docs.python.org/3/reference/datamodel.html#object.__getattr__>)`

or

`[__getattribute__()]`  
`(<https://docs.python.org/3/reference/datamodel.html#object.__getattribute__>)`

function to customize the way

`[dir()]`  
`(<https://docs.python.org/3/library/functions.html#dir>)` reports their attributes. If the object does not provide

`[__dir__()](<https://docs.python.org/3/reference/datamodel.html#object.__dir__>)`, the function tries its best to gather information from the object's

`[__dict__()](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom

`[__getattr__()]`  
`(<https://docs.python.org/3/reference/datamodel.html#object.__getattr__>)`

. The default `[dir()]`  
`(<https://docs.python.org/3/library/functions.html#dir>)` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list

contains the names of its attributes, and recursively of the attributes of its bases. • Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes. The resulting list is sorted alphabetically. For example:>>>

\*\*

```
>>> import** **struct>>>** dir() *# show the names in the module namespace*
['__builtins__', '__name__', 'struct'] *->>>** dir(struct) *# show the names in
the struct module* ['Struct', '__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__initializing__', '__loader__', '__name__', '__package__',
'_clearcache', 'calcsize', 'error', 'pack', 'pack_into', 'unpack', 'unpack_from']
*->>> class** **Shape**: *...* *def* __dir__(self): *...* *return*
['area', 'perimeter', 'location'] *->>>** s = Shape() *->>>** dir(s) ['area',
'location', 'perimeter']
```

**Note** Because `[dir()](<https://docs.python.org/3/library/functions.html#dir>)` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

**divmod** (*a, b*) Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

**enumerate** (*iterable, start=0*) Return an enumerate object. *iterable* must be a sequence, an iterator, or some other object which supports iteration. The

`[__next__()](<https://docs.python.org/3/library/stdtypes.html#iterator.\_\_next\_\_>)` method of the iterator returned by

`[enumerate()](<https://docs.python.org/3/library/functions.html#enumerate>)` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.>>>

```
**
```

```
>>>** seasons = ['Spring', 'Summer', 'Fall', 'Winter'] **>>>**
list(enumerate(seasons)) [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3,
'Winter')] **>>>** list(enumerate(seasons, start=1)) [(1, 'Spring'), (2,
'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
**
```

```
def** enumerate(sequence, start=0): n = start **for** elem **in** sequence:
yield n, elem n += 1
```

**eval** (*expression*[, *globals*[, *locals*]]) The arguments are a string and optional *globals* and *locals*. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object. The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module

`[builtins](<https://docs.python.org/3/library/builtins.html#module-builtins>)` is inserted under that key before *expression* is parsed. This means that *expression* normally has full access to the standard

`[builtins](<https://docs.python.org/3/library/builtins.html#module-builtins>)` module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed with the *globals* and *locals* in the environment where

`[eval()](<https://docs.python.org/3/library/functions.html#eval>)` is called. Note, `eval()` does not have access to the nested scopes (non-locals) in the enclosing environment. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:>>>

```
** >>>** x = 1 **>>>** eval('x+1') 2
```

This function can also be used to execute arbitrary code objects (such as those created by `[compile()](<https://docs.python.org/3/library/functions.html#compile>)`). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the *mode* argument,

`[eval()](<https://docs.python.org/3/library/functions.html#eval>)`'s return value will be `None`. Hints: dynamic execution of statements is supported by the `[exec()](<https://docs.python.org/3/library/functions.html#exec>)` function. The

`[globals()](<https://docs.python.org/3/library/functions.html#globals>)` and  
`[locals()](<https://docs.python.org/3/library/functions.html#locals>)` functions  
returns the current global and local dictionary, respectively, which may be useful to pass around  
for use by `[eval()](<https://docs.python.org/3/library/functions.html#eval>)` or  
`[exec()](<https://docs.python.org/3/library/functions.html#exec>)`. See  
`[ast.literal_eval()]`  
([<https://docs.python.org/3/library/ast.html#ast.literal\\_eval>](https://docs.python.org/3/library/ast.html#ast.literal_eval))

for a function that can safely evaluate strings with expressions containing only literals.

Raises an auditing event `exec` with the code object as the argument. Code compilation events  
may also be raised.

`exec (object[, globals[, locals]])`This function supports dynamic execution of Python code.  
*object* must be either a string or a code object. If it is a string, the string is parsed as a suite of  
Python statements which is then executed (unless a syntax error occurs). 1 If it is a code object,  
it is simply executed. In all cases, the code that's executed is expected to be valid as file input  
(see the section "File input" in the Reference Manual). Be aware that the  
`[nonlocal](<https://docs.python.org/3/reference/simple_stmts.html#nonlocal>)`,  
`[yield](<https://docs.python.org/3/reference/simple_stmts.html#yield>)`, and  
`[return](<https://docs.python.org/3/reference/simple_stmts.html#return>)`  
statements may not be used outside of function definitions even within the context of code  
passed to the `[exec()](<https://docs.python.org/3/library/functions.html#exec>)`  
function. The return value is `None`. In all cases, if the optional parts are omitted, the code is  
executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a  
subclass of dictionary), which will be used for both the global and the local variables. If *globals*  
and *locals* are given, they are used for the global and local variables, respectively. If provided,  
*locals* can be any mapping object. Remember that at module level, *globals* and *locals* are the  
same dictionary. If *exec* gets two separate objects as *globals* and *locals*, the code will be  
executed as if it were embedded in a class definition. If the *globals* dictionary does not contain  
a value for the key `__builtins__`, a reference to the dictionary of the built-in module

`[builtins](<https://docs.python.org/3/library/builtins.html#module-builtins>)` is  
inserted under that key. That way you can control what builtins are available to the executed  
code by inserting your own `__builtins__` dictionary into *globals* before passing it to  
`[exec()](<https://docs.python.org/3/library/functions.html#exec>)`.

Raises an auditing event `exec` with the code object as the argument. Code compilation events  
may also be raised.

## Note The built-in functions

[`globals()`](<<https://docs.python.org/3/library/functions.html#globals>>) and [`locals()`](<<https://docs.python.org/3/library/functions.html#locals>>) return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to

[`exec()`](<<https://docs.python.org/3/library/functions.html#exec>>). **Note** The default `locals` act as described for function

[`locals()`](<<https://docs.python.org/3/library/functions.html#locals>>) below: modifications to the default `locals` dictionary should not be attempted. Pass an explicit `locals` dictionary if you need to see effects of the code on `locals` after function

[`exec()`](<<https://docs.python.org/3/library/functions.html#exec>>) returns.

**filter** (*function, iterable*) Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed. Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if *function* is not `None` and `(item for item in iterable if item)` if *function* is `None`. See [`itertools.filterfalse()`](<<https://docs.python.org/3/library/itertools.html#itertools.filterfalse>>) for the complementary function that returns elements of *iterable* for which *function* returns false.

**class float ([x])** Return a floating point number constructed from a number or string *x*. If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be  `'+'` or  `'-'`; a  `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
**
sign** ::= "+" | "-"
infinity** ::= "Infinity" | "inf"
nan** ::= "nan"
numeric_value ::= [floatnumber]
(<https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-floatnumber>) | [infinity]
(<https://docs.python.org/3/library/functions.html#grammar-token-infinity>) |
[nan](<https://docs.python.org/3/library/functions.html#grammar-token-nan>)
numeric_string ::= [[sign]
(<https://docs.python.org/3/library/string.html#grammar-token-sign>)]
[numeric_value](<https://docs.python.org/3/library/functions.html#grammar-token-numeric-value>)
```

Here `floatnumber` is the form of a Python floating-point literal, described in Floating point literals. Case is not significant, so, for example, “`inf`”, “`Inf`”, “`INFINITY`” and “`iNfINity`” are all acceptable spellings for positive infinity. Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an

```
[OverflowError](<https://docs.python.org/3/library/exceptions.html#OverflowError>)
will be raised. For a general Python object x, float(x) delegates to x.__float__(). If __float__() is not defined then it falls back to
```

```
[__index__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

. If no argument is given, `0.0` is returned. Examples:>>>

```
**
>>>* float('+1.23') 1.23 *>>>* float(' -12345**\\n**') -12345.0 *>>>*
float('1e-003') 0.001 *>>>* float('+1E6') 1000000.0 *>>>* float('-Infinity') -
inf
```

The float type is described in Numeric Types – int, float, complex. *Changed in version 3.6:* Grouping digits with underscores as in code literals is allowed. *Changed in version 3.7:* `x` is now a positional-only parameter. *Changed in version 3.8:* Falls back to

```
[__index__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

if

```
[__float__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__float__>)
```

is not defined.

`format (value[format_spec])` Convert a `value` to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the `value` argument,

however there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language. The default `format_spec` is an empty string which usually gives the same effect as calling

`[str(value)](<https://docs.python.org/3/library/stdtypes.html#str>)` . A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value's

`[__format__()]`  
([https://docs.python.org/3/reference/datamodel.html#object.\\_\\_format\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__format__))

method. A

`[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>)` exception is raised if the method search reaches

`[object](<https://docs.python.org/3/library/functions.html#object>)` and the `format_spec` is non-empty, or if either the `format_spec` or the return value are not strings.

*Changed in version 3.4:* `object().__format__(format_spec)` raises

`[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>)` if `format_spec` is not an empty string.

`class frozenset ([iterable])`Return a new

`[frozenset](<https://docs.python.org/3/library/stdtypes.html#frozenset>)` object, optionally with elements taken from `iterable`. `frozenset` is a built-in class. See

`[frozenset](<https://docs.python.org/3/library/stdtypes.html#frozenset>)` and Set Types – set, frozenset for documentation about this class. For other containers see the built-in

`[set](<https://docs.python.org/3/library/stdtypes.html#set>)` ,  
`[list](<https://docs.python.org/3/library/stdtypes.html#list>)` ,  
`[tuple](<https://docs.python.org/3/library/stdtypes.html#tuple>)` , and  
`[dict](<https://docs.python.org/3/library/stdtypes.html#dict>)` classes, as well as the  
`[collections](<https://docs.python.org/3/library/collections.html#module-collections>)`

module.

`getattr (object, name[, default])`Return the value of the named attribute of `object`. `name` must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x foobar` . If the named attribute does not exist, `default` is returned if provided, otherwise

`[AttributeError]`  
(<https://docs.python.org/3/library/exceptions.html#AttributeError>)

is raised. **Note** Since private name mangling happens at compilation time, one must manually

mangle a private attribute's (attributes with two leading underscores) name in order to retrieve it with `[getattr()](<https://docs.python.org/3/library/functions.html#getattr>)`.

**globals** ()Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

**hasattr** (*object, name*)The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `[AttributeError]` ([<https://docs.python.org/3/library/exceptions.html#AttributeError>](https://docs.python.org/3/library/exceptions.html#AttributeError)) or not.)

**hash** (*object*)Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0). **Note** For objects with custom

`[__hash__()](<https://docs.python.org/3/reference/datamodel.html#object.__hash__>)` methods, note that `[hash()](<https://docs.python.org/3/library/functions.html#hash>)` truncates the return value based on the bit width of the host machine. See `[__hash__()](<https://docs.python.org/3/reference/datamodel.html#object.__hash__>)` for details.

**help** ([*object*])Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated. Note that if a slash(/) appears in the parameter list of a function, when invoking

`[help()](<https://docs.python.org/3/library/functions.html#help>)`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters. This function is added to the built-in namespace by the

`[site](<https://docs.python.org/3/library/site.html#module-site>)` module. *Changed in version 3.4:* Changes to

`[pydoc](<https://docs.python.org/3/library/pydoc.html#module-pydoc>)` and `[inspect](<https://docs.python.org/3/library/inspect.html#module-inspect>)` mean that the reported signatures for callables are now more comprehensive and consistent.

`hex (x)`Convert an integer number to a lowercase hexadecimal string prefixed with “0x”. If `x` is not a Python `[int](<https://docs.python.org/3/library/functions.html#int>)` object, it has to define an

```
[__index__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

method that returns an integer. Some examples:>>>

```
** >>> hex(255) '0xff' >>> hex(-42) '-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:>>>

```
**
```

```
>>> *%#x* % 255, '*%x*' % 255, '*%X*' % 255 ('0xff', 'ff', 'FF') >>>
format(255, '#x'), format(255, 'x'), format(255, 'X') ('0xff', 'ff', 'FF') >>>
f'*{*255*:##x*}', f'*{*255*:x*}', f'*{*255*:X*}' ('0xff', 'ff', 'FF')
```

See also `[format()](<https://docs.python.org/3/library/functions.html#format>)` for more information. See also

`[int()](<https://docs.python.org/3/library/functions.html#int>)` for converting a hexadecimal string to an integer using a base of 16. **Note** To obtain a hexadecimal string representation for a float, use the

```
[float.hex()](<https://docs.python.org/3/library/stdtypes.html#float.hex>)
```

 method.

`id (object)`Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same

`[id()](<https://docs.python.org/3/library/functions.html#id>)` value. **CPython implementation detail:** This is the address of the object in memory. Raises an auditing event `builtins.id` with argument `id`.

`input ([prompt])`If the `prompt` argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read,

```
[EOFError](<https://docs.python.org/3/library/exceptions.html#EOFError>)
```

 is raised.

Example:>>>

\*\*

```
>>>** s = input('--> ') --> Monty Python's Flying Circus **>>>** s "Monty Python's Flying Circus"
```

If the `[readline](<https://docs.python.org/3/library/readline.html#module-readline>)` module was loaded, then

`[input()](<https://docs.python.org/3/library/functions.html#input>)` will use it to provide elaborate line editing and history features.

Raises an auditing event `builtins.input` with argument `prompt` before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

`class int ([x])class int (x, base=10)` Return an integer object constructed from a number or string `x`, or return `0` if no arguments are given. If `x` defines

```
[__int__]()(<https://docs.python.org/3/reference/datamodel.html#object.__int__>) ,
int(x) returns x.__int__(). If x defines
[__index__]()
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
, it returns x.__index__(). If x defines
```

```
[__trunc__]()
(<https://docs.python.org/3/reference/datamodel.html#object.__trunc__>)
, it returns x.__trunc__(). For floating point numbers, this truncates towards zero. If x is not a number or if base is given, then x must be a string,
```

`[bytes](<https://docs.python.org/3/library/stdtypes.html#bytes>)` , or  
`[bytearray](<https://docs.python.org/3/library/stdtypes.html#bytearray>)` instance representing an integer literal in radix `base`. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `z`) having values 10 to 35. The default `base` is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b` / `0B` , `0o` / `0O` , or `0x` / `0X` , as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)` . The integer type is described in Numeric Types – `int`, `float`, `complex`. *Changed in version 3.4:* If `base` is not an instance of `[int](<https://docs.python.org/3/library/functions.html#int>)` and the `base` object has a

```
[base.__index__]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

method, that method is called to obtain an integer for the base. Previous versions used

```
[base.__int__]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__int__>)
```

instead of

```
[base.__index__]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

. *Changed in version 3.6:* Grouping digits with underscores as in code literals is allowed.

*Changed in version 3.7:* *x* is now a positional-only parameter. *Changed in version 3.8:* Falls

back to

```
[__index__()]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

if `__int__()`([<https://docs.python.org/3/reference/datamodel.html#object.\\_\\_int\\_\\_>](https://docs.python.org/3/reference/datamodel.html#object.__int__))

is not defined.

**isinstance** (*object*, *classinfo*)Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or virtual) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples), return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a

```
[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>)
```

exception is raised.

**issubclass** (*class*, *classinfo*)Return `True` if *class* is a subclass (direct, indirect or virtual) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a

```
[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>)
```

exception is raised.

**iter** (*object*[, *sentinel*])Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the iteration protocol (the

```
[__iter__()](<https://docs.python.org/3/reference/datamodel.html#object.__iter__>)
```

method), or it must support the sequence protocol (the

```
[__getitem__()]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__getitem__>)
```

method with integer arguments starting at `0`). If it does not support either of those protocols,

[`TypeError`](<<https://docs.python.org/3/library/exceptions.html#TypeError>>) is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its

[`__next__()`](<[https://docs.python.org/3/library/stdtypes.html#iterator.\\_\\_next\\_\\_](https://docs.python.org/3/library/stdtypes.html#iterator.__next__)>) method; if the value returned is equal to *sentinel*,

[`StopIteration`](<<https://docs.python.org/3/library/exceptions.html#StopIteration>>) will be raised, otherwise the value will be returned. See also [Iterator Types](#). One useful application of the second form of

[`iter()`](<<https://docs.python.org/3/library/functions.html#iter>>) is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

\*\*

```
from** **functools** **import** partial **with** open('mydata.db', 'rb') **as** f:
for block **in** iter(partial(f.read, 64), b''): process_block(block)
```

`len` (*s*)Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set). **C**Python implementation detail: `len` raises

[`OverflowError`](<<https://docs.python.org/3/library/exceptions.html#OverflowError>>) on lengths larger than

[`sys.maxsize`](<<https://docs.python.org/3/library/sys.html#sys.maxsize>>), such as  
[`range(2 ** 100)`](<<https://docs.python.org/3/library/stdtypes.html#range>>).

`class list ([iterable])`Rather than being a function,

[`list`](<<https://docs.python.org/3/library/stdtypes.html#list>>) is actually a mutable sequence type, as documented in [Lists and Sequence Types – list, tuple, range](#).

`locals ()`Update and return a dictionary representing the current local symbol table. Free variables are returned by

[`locals()`](<<https://docs.python.org/3/library/functions.html#locals>>) when it is called in function blocks, but not in class blocks. Note that at the module level,

[`locals()`](<<https://docs.python.org/3/library/functions.html#locals>>) and

[`globals()`](<<https://docs.python.org/3/library/functions.html#globals>>) are the same dictionary. **Note** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

**map** (*function*, *iterable*, ...) Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see

[`itertools.starmap()`]

([<https://docs.python.org/3/library/itertools.html#itertools.starmap>](https://docs.python.org/3/library/itertools.html#itertools.starmap))

**max** (*iterable*, [*key*, *default*]) `max` \*(*arg1*, *arg2*, \**args*[, *key*]) Return the largest item in an iterable or the largest of two or more arguments. If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned. There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for

[`list.sort()`] ([<https://docs.python.org/3/library/stdtypes.html#list.sort>](https://docs.python.org/3/library/stdtypes.html#list.sort)) . The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a

[`ValueError`] ([<https://docs.python.org/3/library/exceptions.html#ValueError>](https://docs.python.org/3/library/exceptions.html#ValueError)) is raised. If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as

`sorted(iterable, key=keyfunc, reverse=True)[0]` and

`heapq.nlargest(1, iterable, key=keyfunc)` . *New in version 3.4:* The *default* keyword-only argument. *Changed in version 3.8:* The *key* can be `None` .

**class memoryview** (*object*) Return a “memory view” object created from the given argument.

See Memory Views for more information.

**min** (*iterable*, [*key*, *default*]) `min` \*(*arg1*, *arg2*, \**args*[, *key*]) Return the smallest item in an iterable or the smallest of two or more arguments. If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned. There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for

[`list.sort()`] ([<https://docs.python.org/3/library/stdtypes.html#list.sort>](https://docs.python.org/3/library/stdtypes.html#list.sort)) . The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a

[`ValueError`] ([<https://docs.python.org/3/library/exceptions.html#ValueError>](https://docs.python.org/3/library/exceptions.html#ValueError)) is

raised. If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as

```
sorted(iterable, key=keyfunc)[0] and heapq.nsmallest(1, iterable, key=keyfunc) .
```

*New in version 3.4:* The `default` keyword-only argument. *Changed in version 3.8:* The `key` can be `None` .

**next** (*iterator*[, *default*]) Retrieve the next item from the *iterator* by calling its

```
[__next__]()(<https://docs.python.org/3/library/stdtypes.html#iterator.__next__>)
```

method. If *default* is given, it is returned if the iterator is exhausted, otherwise

```
[StopIteration](<https://docs.python.org/3/library/exceptions.html#StopIteration>)
```

is raised.

**class\*\* object \*\***Return a new featureless object.

`[object]`(<https://docs.python.org/3/library/functions.html#object>) is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments. **Note**

```
[object](<https://docs.python.org/3/library/functions.html#object>) does not have a
```

```
[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>), so
```

you can't assign arbitrary attributes to an instance of the

```
[object](<https://docs.python.org/3/library/functions.html#object>) class.
```

**oct** (*x*) Convert an integer number to an octal string prefixed with "0o". The result is a valid Python expression. If *x* is not a Python

`[int]`(<https://docs.python.org/3/library/functions.html#int>) object, it has to define an

```
[__index__()]
```

```
(<https://docs.python.org/3/reference/datamodel.html#object.__index__>)
```

method that returns an integer. For example:>>>

```
** >>>** oct(8) '0o10' **>>>** oct(-56) '-0o70'
```

If you want to convert an integer number to octal string either with prefix "0o" or not, you can use either of the following ways.>>>

```
**
```

```
>>>** '*%#o*' % 10, '*%o*' % 10 ('0o12', '12') **>>>** format(10, '#o'),
format(10, 'o') ('0o12', '12') **>>>** f'*{*10*:##o*}*', f'*{*10*:o*}*' ('0o12',
'12')
```

See also `[format()](<https://docs.python.org/3/library/functions.html#format>)` for more information.

`open (file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)` Open `file` and return a corresponding file object. If the file cannot be opened, an `[OSError](<https://docs.python.org/3/library/exceptions.html#OSError>)` is raised. See Reading and Writing Files for more examples of how to use this function. `file` is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless `closefd` is set to `False`.) `mode` is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if `encoding` is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave `encoding` unspecified.) The available modes are:

Character Meaning	<code>'r'</code> open for reading (default)	<code>'w'</code> open for writing, truncating the file first	<code>'x'</code> open for exclusive creation, failing if the file already exists
<code>'a'</code>	open for writing, appending to the end of the file if it exists	<code>'b'</code> binary mode	<code>'t'</code> text mode (default)
<code>'+'</code>	open for updating (reading and writing)	The default mode is <code>'r'</code> (open for reading text, synonym of <code>'rt'</code> ).	<code>'r'</code> (open for reading text, synonym of <code>'rt'</code> )
Modes <code>'w+'</code> and <code>'w+b'</code>	open and truncate the file.		
Modes <code>'r+'</code> and <code>'r+b'</code>	open the file with no truncation.	As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including <code>'b'</code> in the <code>mode</code> argument) return contents as	

`[bytes](<https://docs.python.org/3/library/stdtypes.html#bytes>)` objects without any decoding. In text mode (the default, or when `'t'` is included in the `mode` argument), the contents of the file are returned as

`[str](<https://docs.python.org/3/library/stdtypes.html#str>)`, the bytes having been first decoded using a platform-dependent encoding or using the specified `encoding` if given. There is an additional mode character permitted, `'U'`, which no longer has any effect, and is considered deprecated. It previously enabled universal newlines in text mode, which became the default behaviour in Python 3.0. Refer to the documentation of the `newline` parameter for further details. **Note** Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent. `buffering` is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an

integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on

```
[io.DEFAULT_BUFFER_SIZE]
```

([https://docs.python.org/3/library/io.html#io.DEFAULT\\_BUFFER\\_SIZE](https://docs.python.org/3/library/io.html#io.DEFAULT_BUFFER_SIZE))

. On many systems, the buffer will typically be 4096 or 8192 bytes long.

- “Interactive” text files (files for which

```
[isatty]()
```

(<https://docs.python.org/3/library/io.html#io.IOBase.isatty>) returns True ) use line buffering. Other text files use the policy described above for binary files.

*encoding* is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever

```
[locale.getpreferredencoding()]
```

(<https://docs.python.org/3/library/locale.html#locale.getpreferredencoding>)

returns), but any text encoding supported by Python can be used. See the

```
[codecs]()
```

(<https://docs.python.org/3/library/codecs.html#module-codecs>) module for the list of supported encodings. *errors* is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers), though any error handling name that has been registered with

```
[codecs.register_error()]
```

([https://docs.python.org/3/library/codecs.html#codecs.register\\_error](https://docs.python.org/3/library/codecs.html#codecs.register_error))

is also valid. The standard names include:

- 'strict' to raise a

```
[ValueError]()
```

(<https://docs.python.org/3/library/exceptions.html#ValueError>)

exception if there is an encoding error. The default value of `None` has the same effect.

- 'ignore' ignores errors. Note that ignoring encoding errors can lead to data loss.

- 'replace' causes a replacement marker (such as '?' ) to be inserted where there is malformed data.
- 'surrogateescape' will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.

- 'xmlcharrefreplace' is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.

- 'backslashreplace' replaces malformed data by Python’s backslashed escape sequences.

- 'namereplace' (also only supported when writing) replaces unsupported characters with `\N{...}` escape sequences. *newline* controls how universal newlines mode works (it only applies to text mode). It can be `None`, '', '`\n`', '`\r`', and '`\r\n`'. It works as

follows:

- When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `'\\n'`, `'\\r'`, or `'\\r\\n'`, and these are translated into `'\\n'` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if `newline` is `None`, any `'\\n'` characters written are translated to the system default line separator,

`[os.linesep](<https://docs.python.org/3/library/os.html#os.linesep>)` . If `newline` is `''` or `'\\n'`, no translation takes place. If `newline` is any of the other legal values, any `'\\n'` characters written are translated to the given string. If `closefd` is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` must be `True` (the default) otherwise an error will be raised. A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing

`[os.open](<https://docs.python.org/3/library/os.html#os.open>)` as `opener` results in functionality similar to passing `None`). The newly created file is non-inheritable. The following example uses the `dir_fd` parameter of the

`[os.open()](<https://docs.python.org/3/library/os.html#os.open>)` function to open a file relative to a given directory:>>>

\*\*

```
>>> import** **os>>>** dir_fd = os.open('somedir', os.O_RDONLY) **>>> def**
opener(path, flags): **....** **return** os.open(path, flags, dir_fd=dir_fd)
....>>> with open('spamspam.txt', 'w', opener=opener) **as** f: **....**
print('This will be written to somedir/spamspam.txt', file=f) **...>>>**
os.close(dir_fd) *# don't leak a file descriptor*
```

The type of file object returned by the

`[open()](<https://docs.python.org/3/library/functions.html#open>)` function depends on the mode. When

`[open()](<https://docs.python.org/3/library/functions.html#open>)` is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a subclass of

`[io.TextIOWrapper](<https://docs.python.org/3/library/io.html#io.TextIOWrapper>)` (specifically

`[io.TextIOWrapper](<https://docs.python.org/3/library/io.html#io.TextIOWrapper>)`).

When used to open a file in a binary mode with buffering, the returned class is a subclass of

`[io.BufferedIOBase](<https://docs.python.org/3/library/io.html#io.BufferedIOBase>)`

. The exact class varies: in read binary mode, it returns an

[`io.BufferedReader`](<<https://docs.python.org/3/library/io.html#io.BufferedReader>>)  
; in write binary and append binary modes, it returns an  
[`io.BufferedWriter`](<<https://docs.python.org/3/library/io.html#io.BufferedWriter>>)  
, and in read/write mode, it returns an  
[`io.BufferedRandom`](<<https://docs.python.org/3/library/io.html#io.BufferedRandom>>)  
. When buffering is disabled, the raw stream, a subclass of  
[`io.RawIOBase`](<<https://docs.python.org/3/library/io.html#io.RawIOBase>>),  
[`io.FileIO`](<<https://docs.python.org/3/library/io.html#io.FileIO>>), is returned. See  
also the file handling modules, such as,  
[`fileinput`](<<https://docs.python.org/3/library/fileinput.html#module-fileinput>>),  
[`io`](<<https://docs.python.org/3/library/io.html#module-io>>) (where  
[`open()`](<<https://docs.python.org/3/library/functions.html#open>>) is declared),  
[`os`](<<https://docs.python.org/3/library/os.html#module-os>>),  
[`os.path`](<<https://docs.python.org/3/library/os.path.html#module-os.path>>),  
[`tempfile`](<<https://docs.python.org/3/library/tempfile.html#module-tempfile>>), and  
[`shutil`](<<https://docs.python.org/3/library/shutil.html#module-shutil>>). Raises an  
auditing event `open` with arguments `file`, `mode`, `flags`. The `mode` and `flags`  
arguments may have been modified or inferred from the original call.*Changed in version 3.3:* •  
The `opener` parameter was added. • The '`x`' mode was added. •  
[`IOError`](<<https://docs.python.org/3/library/exceptions.html#IOError>>) used to be  
raised, it is now an alias of  
[`OSError`](<<https://docs.python.org/3/library/exceptions.html#OSError>>). •  
[`FileExistsError`]  
(<<https://docs.python.org/3/library/exceptions.html#FileExistsError>>)  
is now raised if the file opened in exclusive creation mode ('`x`') already exists.*Changed in  
version 3.4:* • The file is now non-inheritable. *Deprecated since version 3.4, will be removed in  
version 3.10:* The '`u`' mode.*Changed in version 3.5:* • If the system call is interrupted and the  
signal handler does not raise an exception, the function now retries the system call instead of  
raising an  
[`InterruptedError`]  
(<<https://docs.python.org/3/library/exceptions.html#InterruptedError>>)  
exception (see **PEP 475** for the rationale). • The '`namereplace`' error handler was  
added.*Changed in version 3.6:* • Support added to accept objects implementing  
[`os.PathLike`](<<https://docs.python.org/3/library/os.html#os.PathLike>>). • On  
Windows, opening a console buffer may return a subclass of  
[`io.RawIOBase`](<<https://docs.python.org/3/library/io.html#io.RawIOBase>>) other than  
[`io.FileIO`](<<https://docs.python.org/3/library/io.html#io.FileIO>>).

**ord** (*c*) Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of  
`[chr()](<https://docs.python.org/3/library/functions.html#chr>)`.

**pow** (*base, exp[, mod]*) Return *base* to the power *exp*; if *mod* is present, return *base* to the power *exp*, modulo *mod* (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`. The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `[int](<https://docs.python.org/3/library/functions.html#int>)` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. For `[int](<https://docs.python.org/3/library/functions.html#int>)` operands *base* and *exp*, if *mod* is present, *mod* must also be of integer type and *mod* must be nonzero. If *mod* is present and *exp* is negative, *base* must be relatively prime to *mod*. In that case, `pow(inv_base, -exp, mod)` is returned, where *inv\_base* is an inverse to *base* modulo *mod*. Here's an example of computing an inverse for `38` modulo `97`:>>>

```
** >>>** pow(38, -1, mod=97) 23 **>>>** 23 * 38 % 97 == 1 True
```

*Changed in version 3.8:* For

`[int](<https://docs.python.org/3/library/functions.html#int>)` operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses. *Changed in version 3.8:* Allow keyword arguments. Formerly, only positional arguments were supported.

**print** (\**objects*, *sep*='', *end*='\n', *file*=`sys.stdout`, *flush*=`False`) Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments. All non-keyword arguments are converted to strings like

`[str()](<https://docs.python.org/3/library/stdtypes.html#str>)` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given,

`[print()](<https://docs.python.org/3/library/functions.html#print>)` will just write *end*. The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `[sys.stdout](<https://docs.python.org/3/library/sys.html#sys.stdout>)` will be used. Since printed arguments are converted to text strings,

[print()](<<https://docs.python.org/3/library/functions.html#print>>) cannot be used with binary mode file objects. For these, use `file.write(...)` instead. Whether output is buffered is usually determined by `file`, but if the `flush` keyword argument is true, the stream is forcibly flushed. *Changed in version 3.3:* Added the `flush` keyword argument.

`class property (fget=None, fset=None, fdel=None, doc=None)`Return a property attribute. `fget` is a function for getting an attribute value. `fset` is a function for setting an attribute value. `fdel` is a function for deleting an attribute value. And `doc` creates a docstring for the attribute. A typical use is to define a managed attribute `x`:

```
`class C: def init(self): self._x = Nonedef getx(self): return self._x
```

```
1 **def** setx(self, value):
2 self._x = value
3
4 **def** delx(self):
5 **del** self._x
6
7 x = property(getx, setx, delx, "I'm the 'x' property.")`
```

If `c` is an instance of `C`, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter. If given, `doc` will be the docstring of the property attribute. Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using

[`property()`](<<https://docs.python.org/3/library/functions.html#property>>) as a decorator:

```
`class Parrot: def init(self): self._voltage = 100000
```

```
1 **@propertydef** voltage(self):
2 """Get the current voltage."""***return** self._voltage`
```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for `voltage` to “Get the current voltage.” A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy

of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
`class C: def init(self): self._x = None@propertydef x(self): *"""\nI'm the 'x' property.\n"""\nreturn\nself._x
```

```
1 **@x**.setter\n2 **def** x(self, value):\n3 self._x = value\n4\n5 **@x**.deleter\n6 **def** x(self):\n7 **del** self._x`
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.) The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments. *Changed in version 3.5:* The docstrings of property objects are now writeable.

`class range (stop)class range (start, stop[, step])`Rather than being a function, `[range](<https://docs.python.org/3/library/stdtypes.html#range>)` is actually an immutable sequence type, as documented in Ranges and Sequence Types – list, tuple, range.

`repr (object)`Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `[eval()](<https://docs.python.org/3/library/functions.html#eval>)`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a

`[__repr__()](<https://docs.python.org/3/reference/datamodel.html#object.__repr__>)` method.

`reversed (seq)`Return a reverse iterator. `seq` must be an object which has a

`[__reversed__()](<https://docs.python.org/3/reference/datamodel.html#object.__reversed__>)`

method or supports the sequence protocol (the

`[__len__()](<https://docs.python.org/3/reference/datamodel.html#object.__len__>)`

method and the

```
[__getitem__()]
(<https://docs.python.org/3/reference/datamodel.html#object.__getitem__>)
```

method with integer arguments starting at `0` ).

`round` (*number*[, *ndigits*]) Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input. For the built-in types supporting `[round()]` (<https://docs.python.org/3/library/functions.html#round>) , values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise the return value has the same type as *number*. For a general Python object `number` , `round` delegates to `number.__round__` . **Note** The behavior of

`[round()]` (<https://docs.python.org/3/library/functions.html#round>) for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68` . This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating Point Arithmetic: Issues and Limitations for more information.

`class set ([iterable])` Return a new

`[set](<https://docs.python.org/3/library/stdtypes.html#set>)` object, optionally with elements taken from *iterable*. `set` is a built-in class. See

`[set](<https://docs.python.org/3/library/stdtypes.html#set>)` and Set Types – `set`, `frozenset` for documentation about this class. For other containers see the built-in

```
[frozenset](<https://docs.python.org/3/library/stdtypes.html#frozenset>),
[list](<https://docs.python.org/3/library/stdtypes.html#list>),
[tuple](<https://docs.python.org/3/library/stdtypes.html#tuple>), and
[dict](<https://docs.python.org/3/library/stdtypes.html#dict>)
[collections](<https://docs.python.org/3/library/collections.html#module-collections>)
```

module.

`setattr` (*object*, *name*, *value*) This is the counterpart of

`[getattr()]` (<https://docs.python.org/3/library/functions.html#getattr>) . The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123` . **Note**

Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with

```
[setattr()](<https://docs.python.org/3/library/functions.html#setattr>) .
```

`class slice (stop)``class slice (start, stop[, step])`Return a slice object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See

```
[itertools.islice()](<https://docs.python.org/3/library/itertools.html#itertools.islice>)
```

for an alternate version that returns an iterator.

`sorted (iterable, ***, key=None, reverse=False)`Return a new sorted list from the items in `iterable`. Has two optional arguments which must be specified as keyword arguments. `key` specifies a function of one argument that is used to extract a comparison key from each element in `iterable` (for example, `key=str.lower`). The default value is `None` (compare the elements directly). `reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed. Use

```
[functools.cmp_to_key()](<https://docs.python.org/3/library/functools.html#functools.cmp_to_key>)
```

to convert an old-style `cmp` function to a `key` function. The built-in

`[sorted()](<https://docs.python.org/3/library/functions.html#sorted>)` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal – this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade). For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

`@**staticmethod**` Transform a method into a static method. A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
** class** **C**: **@staticmethoddef** f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function decorator – see Function definitions for details. A static method can be called either on the class (such as `c.f()`) or on an instance (such as `c().f()`). Static methods in Python are similar to those found in Java or C++. Also see

[`classmethod()`](<<https://docs.python.org/3/library/functions.html#classmethod>>) for a variant that is useful for creating alternate class constructors. Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
** class** **C**: builtin_open = staticmethod(open)
```

For more information on static methods, see [The standard type hierarchy](#).

`class str (object=")class str (object=b", encoding='utf-8', errors='strict')`Return a [str](<<https://docs.python.org/3/library/stdtypes.html#str>>) version of *object*. See [str()](<<https://docs.python.org/3/library/stdtypes.html#str>>) for details. `str` is the built-in string class. For general information about strings, see [Text Sequence Type – str](#).

`sum (iterable, /, start=0)`Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string. For some use cases, there are good alternatives to

[`sum()`](<<https://docs.python.org/3/library/functions.html#sum>>). The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see

[`math.fsum()`](<<https://docs.python.org/3/library/math.html#math.fsum>>). To concatenate a series of iterables, consider using

```
[itertools.chain()]
(<https://docs.python.org/3/library/itertools.html#itertools.chain>)
```

. *Changed in version 3.8*: The *start* parameter can be specified as a keyword argument.

`super ([type| object-or-type])`Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class. The *object-or-type* determines the method resolution order to be searched. The search starts from the class right after the *type*. For example, if

[`__mro__`](<[https://docs.python.org/3/library/stdtypes.html#class.\\_\\_mro\\_\\_](https://docs.python.org/3/library/stdtypes.html#class.__mro__)>) of *object-or-type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then

[`super()`](<<https://docs.python.org/3/library/functions.html#super>>) searches `C -> A -> object`. The

[`__mro__`](<[https://docs.python.org/3/library/stdtypes.html#class.\\_\\_mro\\_\\_](https://docs.python.org/3/library/stdtypes.html#class.__mro__)>) attribute of the *object-or-type* lists the method resolution search order used by both

`[getattr()]`(<https://docs.python.org/3/library/functions.html#getattr>) and `[super()]`(<https://docs.python.org/3/library/functions.html#super>) . The attribute is dynamic and can change whenever the inheritance hierarchy is updated. If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods). There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages. The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime). For both use cases, a typical superclass call looks like this:

\*\*

```
class** **C***(B): **def** method(self, arg): super().method(arg) *# This does the
same thing as:# super(C, self).method(arg)*
```

In addition to method lookups,

`[super()]`(<https://docs.python.org/3/library/functions.html#super>) also works for attribute lookups. One possible use case for this is calling descriptors in a parent or sibling class. Note that `[super()]`(<https://docs.python.org/3/library/functions.html#super>) is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)` . It does so by implementing its own

```
[__getattribute__()
(https://docs.python.org/3/reference/datamodel.html#object.__getattribute__)
```

method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly,

`[super()]`(<https://docs.python.org/3/library/functions.html#super>) is undefined for implicit lookups using statements or operators such as `super()[name]` . Also note that, aside from the zero argument form,

`[super()]`(<https://docs.python.org/3/library/functions.html#super>) is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as

accessing the current instance for ordinary methods. For practical suggestions on how to design cooperative classes using

[super()](<<https://docs.python.org/3/library/functions.html#super>>) , see guide to using super().

`class tuple ([iterable])`Rather than being a function,

[tuple](<<https://docs.python.org/3/library/stdtypes.html#tuple>>) is actually an immutable sequence type, as documented in Tuples and Sequence Types – list, tuple, range.

`class type (object)``class type (name, bases, dict, **kwds)`With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by

[object.\_\_class\_\_]  
(<[https://docs.python.org/3/library/stdtypes.html#instance.\\_\\_class\\_\\_](https://docs.python.org/3/library/stdtypes.html#instance.__class__)>)

. The [isinstance()](<<https://docs.python.org/3/library/functions.html#isinstance>>) built-in function is recommended for testing the type of an object, because it takes subclasses into account. With three arguments, return a new type object. This is essentially a dynamic form of the

[class](<[https://docs.python.org/3/reference/compound\\_stmts.html#class](https://docs.python.org/3/reference/compound_stmts.html#class)>) statement.

The *name* string is the class name and becomes the

[\_\_name\_\_](<[https://docs.python.org/3/library/stdtypes.html#define\\_type\\_name](https://docs.python.org/3/library/stdtypes.html#define_type_name)>) attribute. The *bases* tuple contains the base classes and becomes the

[\_\_bases\_\_](<[https://docs.python.org/3/library/stdtypes.html#class.\\_\\_bases\\_\\_](https://docs.python.org/3/library/stdtypes.html#class.__bases__)>) attribute; if empty,

[object](<<https://docs.python.org/3/library/functions.html#object>>) , the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the

[\_\_dict\_\_](<[https://docs.python.org/3/library/stdtypes.html#object.\\_\\_dict\\_\\_](https://docs.python.org/3/library/stdtypes.html#object.__dict__)>) attribute. The following two statements create identical

[type](<<https://docs.python.org/3/library/functions.html#type>>) objects:>>>

```
** >>> class** **X**: **....** a = 1 **...>>>** X = type('X', (), dict(a=1))
```

See also Type Objects. Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually

[\_\_init\_subclass\_\_]  
(<[https://docs.python.org/3/reference/datamodel.html#object.\\_\\_init\\_subclass\\_\\_](https://docs.python.org/3/reference/datamodel.html#object.__init_subclass__)>)

) in the same way that keywords in a class definition (besides *metaclass*) would. See also Customizing class creation. *Changed in version 3.6:* Subclasses of

`[type](<https://docs.python.org/3/library/functions.html#type>)` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

**vars ([object])**Return the

`[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)`

attribute for a module, class, instance, or any other object with a

`[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)`

attribute. Objects such as modules and instances have an updateable

`[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)`

attribute; however, other objects may have write restrictions on their

`[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)`

attributes (for example, classes use a

`[types.MappingProxyType]`

(<https://docs.python.org/3/library/types.html#types.MappingProxyType>)

to prevent direct dictionary updates). Without an argument,

`[vars()](<https://docs.python.org/3/library/functions.html#vars>)` acts like

`[locals()](<https://docs.python.org/3/library/functions.html#locals>)`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored. A

`[TypeError](<https://docs.python.org/3/library/exceptions.html#TypeError>)`

exception is raised if an object is specified but it doesn't have a

`[__dict__](<https://docs.python.org/3/library/stdtypes.html#object.__dict__>)`

attribute (for example, if its class defines the

`[__slots__](<https://docs.python.org/3/reference/datamodel.html#object.__slots__>)`

attribute).

**zip (\*iterables)**Make an iterator that aggregates elements from each of the iterables. Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

\*\*

```
def** zip(*iterables): **# zip('ABCD', 'xy') --> Ax By**sentinel = object()
iterators = [iter(it) **for** it **in** iterables] **while** iterators: result =
[] **for** it **in** iterators: elem = next(it, sentinel) **if** elem **is** sentinel:
returnresult.append(elem) **yield** tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

`[zip()](<https://docs.python.org/3/library/functions.html#zip>)` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use

```
[itertools.zip_longest()]
(<https://docs.python.org/3/library/itertools.html#itertools.zip_longest>)
```

instead. `[zip()](<https://docs.python.org/3/library/functions.html#zip>)` in conjunction with the `*` operator can be used to unzip a list:>>>

\*\*

```
>>>** x = [1, 2, 3] **>>>** y = [4, 5, 6] **>>>** zipped = zip(x, y) **>>>**
list(zipped) [(1, 4), (2, 5), (3, 6)] **>>>** x2, y2 = zip(*zip(x, y)) **>>>** x
== list(x2) **and** y == list(y2) True
```

`__import__` (*name, globals=None, locals=None, fromlist=(), level=0*)**Note** This is an advanced function that is not needed in everyday Python programming, unlike

```
[importlib.import_module()]
(<https://docs.python.org/3/library/importlib.html#importlib.import_module>)
```

. This function is invoked by the

```
[import](<https://docs.python.org/3/reference/simple_stmts.html#import>) statement.
```

It can be replaced (by importing the

```
[builtins](<https://docs.python.org/3/library/builtins.html#module-builtins>)
```

module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of

`[__import__]()(<https://docs.python.org/3/library/functions.html#__import__>)` is also discouraged in favor of

```
[importlib.import_module()]
(<https://docs.python.org/3/library/importlib.html#importlib.import_module>)
```

. The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the

[import](<[https://docs.python.org/3/reference/simple\\_stmts.html#import](https://docs.python.org/3/reference/simple_stmts.html#import)>) statement. *level* specifies whether to use absolute or relative imports. 0 (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling

[\_\_import\_\_]()(<[https://docs.python.org/3/library/functions.html#\\_\\_import\\_\\_](https://docs.python.org/3/library/functions.html#__import__)>) (see PEP 328 for the details). When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned. For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how

[\_\_import\_\_]()(<[https://docs.python.org/3/library/functions.html#\\_\\_import\\_\\_](https://docs.python.org/3/library/functions.html#__import__)>) returns the toplevel module here because this is the object that is bound to a name by the [import](<[https://docs.python.org/3/reference/simple\\_stmts.html#import](https://docs.python.org/3/reference/simple_stmts.html#import)>) statement. On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0) eggs =
_temp.eggs saus = _temp.sausage
```

Here, the `spam.ham` module is returned from

```
[__import__]()(<https://docs.python.org/3/library/functions.html#__import__>).
```

From this object, the names to import are retrieved and assigned to their respective names. If you simply want to import a module (potentially within a package) by name, use

```
[importlib.import_module()
(<https://docs.python.org/3/library/importlib.html#importlib.import_module>)]
```

2021-03-06\_Python-Study-Guide-for-a-JavaScript-Programmer-

Built-in Types

# Aux Resources

<<<<< HEAD

---

## Notes I Wish I Had When I Started Learning Python

Plus resources for learning data structures and algorithms in python at the bottom of this article!



Bryan Guner Aug 24 · 22 min read



## Basics

- **PEP8** : Python Enhancement Proposals, style-guide for Python.
- `print` is the equivalent of `console.log` .
- `#` is used to make comments in your code.

```
1 def foo():
2 """
3 The foo function does many amazing things that you
4 should not question. Just accept that it exists and
5 use it with caution.
6 """
7 secretThing()
```

- Python has a built in help function that let's you see a description of the source code without having to navigate to it.

# Numbers

- Python has three types of numbers:
- **Integer**
- Positive and Negative Counting Numbers.
- No Decimal Point
- Created by a literal non-decimal pt number or with the `int()` constructor.

```
print(3) # => 3 print(int(19)) # => 19 print(int()) # => 0
```

- Boolean is a subtype of integer in Python.
- **Floating Point Number**
- Decimal Numbers.

```
print(2.24) # => 2.24 print(2.) # => 2.0 print(float()) # => 0.0 print(27e-5) #
```

# Complex Numbers

- Consist of a real part and imaginary part.
- The `i` is switched to a `j` in programming.

```
print(7j) # => 7j print(5.1+7.7j) # => 5.1+7.7j print(complex(3, 5)) # => 3+5j
```

## Type Casting : The process of converting one number to another.

```
1 # Using Float
2 print(17) # => 17
3 print(float(17)) # => 17.0# Using Int
4 print(17.0) # => 17.0
5 print(int(17.0)) # => 17# Using Str
6 print(str(17.0) + ' and ' + str(17)) # => 17.0 and 17
```

- The arithmetic operators are the same between JS and Python, with two additions:
- “\*\*” : Double asterisk for exponent.
- “//” : Integer Division.
- There are no spaces between math operations in Python.
- Integer Division gives the other part of the number from Module; it is a way to do round down numbers replacing `Math.floor()` in JS.
- There are no `++` and `--` in Python, the only shorthand operators are:

- `+=` (addition)
- `-=` (subtraction)
- `*=` (multiplication)
- `/=` (division)
- `%=` (modulo)
- `**=` (exponentiation)
- `//=` (integer division)

---

## Strings

- Python uses both single and double quotes.
- You can escape strings like so `'Jodi asked, "What\'s up, Sam?"'`
- Multiline strings use triple quotes.

```
1 print('''My instructions are very long so to make them
2 more readable in the code I am putting them on
3 more than one line. I can even include "quotes"
4 of any kind because they won't get confused with
5 the end of the string!'''')
```

- Use the `len()` function to get the length of a string.

```
print(len("Spaghetti")) # => 9
```

- Python uses zero-based indexing
- Python allows negative indexing (thank god!)

```
print("Spaghetti)[-1]) # => i print("Spaghetti)[-4]) # => e
```

- Python let's you use ranges

```
print("Spaghetti"[1:4]) # => pag print("Spaghetti"[4:-1]) # => hett print("Spag
```

- The end range is exclusive just like slice in JS.

```
1 # Shortcut to get from the beginning of a string to a certain index.
2 print("Spaghetti":4)) # => Spag
3 print("Spaghetti":-1)) # => Spaghett# Shortcut to get from a certain index
4 print("Spaghetti":1)) # => paghetti
5 print("Spaghetti":-4:)) # => etti
```

- The index string function is the equiv. of indexOf() in JS

```
1 print("Spaghetti".index("h")) # => 4
2 print("Spaghetti".index("t")) # => 6
```

- The count function finds out how many times a substring appears in a string.

```
1 print("Spaghetti".count("h")) # => 1
2 print("Spaghetti".count("t")) # => 2
3 print("Spaghetti".count("s")) # => 0
4 print('''We choose to go to the moon in this decade and do the other things,
```

```

5 not because they are easy, but because they are hard, because that goal will
6 serve to organize and measure the best of our energies and skills, because tha
7 challenge is one that we are willing to accept, one we are unwilling to
8 postpone, and one which we intend to win, and the others, too.
9 ''' .count('the ') # => 4

```

- You can use `+` to concatenate strings, just like in JS.
- You can also use `*` to repeat strings or multiply strings.
- Use the `format()` function to use placeholders in a string to input values later on.

```

1 first_name = "Billy"
2 last_name = "Bob"
3 print('Your name is {0} {1}'.format(first_name, last_name)) # => Your name is

```

- Shorthand way to use format function is:

```
print(f'Your name is {first_name} {last_name}')
```

- Some useful string methods.
- Note that in JS `join` is used on an Array, in Python it is used on String.

Value	Method	Result
s = "Hello"	s.upper()	"HELLO"
s = "Hello"	s.lower()	"hello"
s = "Hello"	s.islower()	False
s = "hello"	s.islower()	True
s = "Hello"	s.isupper()	False
s = "HELLO"	s.isupper()	True
s = "Hello"	s.startswith("He")	True
s = "Hello"	s.endswith("lo")	True
s = "Hello World"	s.split()	["Hello", "World"]
s = "i-am-a-dog"	s.split("-")	["i", "am", "a", "dog"]

- There are also many handy testing methods.

Method	Purpose
isalpha()	returns <code>True</code> if the string consists only of letters and is not blank.
isalnum()	returns <code>True</code> if the string consists only of letters and numbers and is not blank.
isdecimal()	returns <code>True</code> if the string consists only of numeric characters and is not blank.
isspace()	returns <code>True</code> if the string consists only of spaces, tabs, and newlines and is not blank.
istitle()	returns <code>True</code> if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

## Variables and Expressions

- **Duck-Typing** : Programming Style which avoids checking an object's type to figure out what it can do.
- Duck Typing is the fundamental approach of Python.
- Assignment of a value automatically declares.

```
1 a = 7
2 b = 'Marbles'
3 print(a) # => 7
4 print(b) # => Marbles
```

- You can chain variable assignments to give multiple var names the same value.
- Use with caution as this is highly unreadable

```
1 count = max = min = 0
2 print(count) # => 0
3 print(max) # => 0
4 print(min) # => 0
```

- The value and type of a variable can be re-assigned at any time.

```
1 a = 17
2 print(a) # => 17
3 a = 'seventeen'
4 print(a) # => seventeen
```

- `NaN` does not exist in Python, but you can 'create' it like so: `print(float("nan"))`
- Python replaces `null` with `None`.
- `None` is an object and can be directly assigned to a variable.
- Using `None` is a convenient way to check to see why an action may not be operating correctly in your program.

---

## Boolean Data Type

- One of the biggest benefits of Python is that it reads more like English than JS does.

Python	JavaScript
and	&&
or	
not	!

```

1 # Logical AND
2 print(True and True) # => True
3 print(True and False) # => False
4 print(False and False) # => False# Logical OR
5 print(True or True) # => True
6 print(True or False) # => True
7 print(False or False) # => False# Logical NOT
8 print(not True) # => False
9 print(not False and True) # => True
10 print(not True or False) # => False

```

- By default, Python considers an object to be true UNLESS it is one of the following:
- Constant `None` or `False`
- Zero of any numeric type.
- Empty Sequence or Collection.
- `True` and `False` must be capitalized

## Comparison Operators

- Python uses all the same equality operators as JS.
- In Python, equality operators are processed from left to right.
- Logical operators are processed in this order:
  - **NOT**
  - **AND**
  - **OR**
- Just like in JS, you can use `parentheses` to change the inherent order of operations.
- **Short Circuit**: Stopping a program when a `true` or `false` has been reached.

Expression	Right side evaluated?
True and ...	Yes
False and ...	No
True or ...	No
False or ...	Yes

## Identity vs Equality

```
1 print (2 == '2') # => False
2 print (2 is '2') # => Falseprint ("2" == '2') # => True
3 print ("2" is '2') # => True# There is a distinction between the number type
4 print (2 == 2.0) # => True
5 print (2 is 2.0) # => False
```

- In the Python community it is better to use `is` and `is not` over `==` or `!=`

## If Statements

```
1 if name == 'Monica':
2 print('Hi, Monica.')
3 print('Hi, Monica.')
4 else:
5 print('Hello, stranger.')
6 print('Hi, Monica.')
7 elif age < 12:
8 print('You are not Monica, kiddo.')
9 elif age > 2000:
10 print('Unlike you, Monica is not an undead, immortal vampire.')
11 elif age > 100:
12 print('You are not Monica, grannie.')
```

- Remember the order of `elif` statements matter.

## While Statements

```
1 spam = 0
2 while spam < 5:
3 print('Hello, world.')
4 spam = spam + 1
```

- `Break` statement also exists in Python.

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam >= 5:
6 break
```

- As are `continue` statements

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam < 5:
6 continue
7 break
```

## Try/Except Statements

- Python equivalent to `try/catch`

```
1 a = 321
2 try:
3 print(len(a))
4 except:
```

```
5 print('Silently handle error here') # Optionally include a correction
6 a = str(a)
7 print(len(a)a = '321'
8 try:
9 print(len(a))
10 except:
11 print('Silently handle error here') # Optionally include a correction
12 a = str(a)
13 print(len(a))
```

- You can name an error to give the output more specificity.

```
1 a = 100
2 b = 0
3 try:
4 c = a / b
5 except ZeroDivisionError:
6 c = None
7 print(c)
```

- You can also use the `pass` command to bypass a certain error.

```
1 a = 100
2 b = 0
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
```

- The `pass` method won't allow you to bypass every single error so you can chain an exception series like so:

```
1 a = 100
2 # b = "5"
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
7 except (TypeError, NameError):
8 print("ERROR!")
```

- You can use an `else` statement to end a chain of `except` statements.

```

1 # tuple of file names
2 files = ('one.txt', 'two.txt', 'three.txt')# simple loop
3 for filename in files:
4 try:
5 # open the file in read mode
6 f = open(filename, 'r')
7 except OSError:
8 # handle the case where file does not exist or permission is denied
9 print('cannot open file', filename)
10 else:
11 # do stuff with the file object (f)
12 print(filename, 'opened successfully')
13 print('found', len(f.readlines()), 'lines')
14 f.close()

```

- `finally` is used at the end to clean up all actions under any circumstance.

```

1 def divide(x, y):
2 try:
3 result = x / y
4 except ZeroDivisionError:
5 print("Cannot divide by zero")
6 else:
7 print("Result is", result)
8 finally:
9 print("Finally...")

```

- Using duck typing to check to see if some value is able to use a certain method.

```

1 # Try a number - nothing will print out
2 a = 321
3 if hasattr(a, '__len__'):
4 print(len(a))# Try a string - the length will print out (4 in this case)
5 b = "5555"
6 if hasattr(b, '__len__'):
7 print(len(b))

```

## Pass

- Pass Keyword is required to write the JS equivalent of :

```
1 if (true) {
2 }while (true) {}if True:
3 passwhile True:
4 pass
```

## Functions

- **Function** definition includes:
- The `def` keyword
- The name of the function
- A list of parameters enclosed in parentheses.
- A colon at the end of the line.
- One tab indentation for the code to run.

```
1 def printCopyright():
2 print("Copyright 2020. Me, myself and I. All rights reserved.")
```

- You can use default parameters just like in JS

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)greeting("Monica")
3 # Hello Monica
4 greeting("Barry", "Hey")
5 # Hey Barry
```

- Keep in mind, default parameters must always come after regular parameters.

```
1 # THIS IS BAD CODE AND WILL NOT RUN
```

```
2 def increment(delta=1, value):
3 return delta + value
```

- You can specify arguments by name without destructuring in Python.

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)# name has no default value, so just provide the value
3 # saying has a default value, so use a keyword argument
4 greeting("Monica", saying="Hi")
```

- The `lambda` keyword is used to create anonymous functions and are supposed to be `one-liners`.

```
toUpper = lambda s: s.upper()
```

## Notes

---

## Formatted Strings

- Remember that in Python `join()` is called on a string with an array/list passed in as the argument.

```
1 shopping_list = ['bread','milk','eggs']
2 print(', '.join(shopping_list))
```

- Python has a very powerful formatting engine.
- `format()` is also applied directly to strings.

```
1 # Comma Thousands Separator
2 print('{:,}'.format(1234567890))
```

```

3 '1,234,567,890'# Date and Time
4 d = datetime.datetime(2020, 7, 4, 12, 15, 58)
5 print(':{%Y-%m-%d %H:%M:%S}'.format(d))
6 '2020-07-04 12:15:58'# Percentage
7 points = 190
8 total = 220
9 print('Correct answers: {:.2%}'.format(points/total))
10 Correct answers: 86.36%# Data Tables
11 width=8
12 print(' decimal hex binary')
13 print('-'*27)
14 for num in range(1,16):
15 for base in 'dXb':
16 print('{0:{width}{base}}'.format(num, base=base, width=width), end='
17 print()

```

## Getting Input from the Command Line

- Python runs synchronously, all programs and processes will stop when listening for a user input.
- The `input` function shows a prompt to a user and waits for them to type 'ENTER'.

## Scripts vs Programs

- **Programming Script** : A set of code that runs in a linear fashion.
- The largest difference between scripts and programs is the level of complexity and purpose. Programs typically have many UI's.
- Python can be used to display html, css, and JS.
- We will be using Python as an API (Application Programming Interface)

## Structured Data

- **Sequence** : The most basic data structure in Python where the index determines the order.
- List
- Tuple

- Range

**Collections** : Unordered data structures, hashable values.

- Dictionaries
  - Sets
  - **Iterable** : Generic name for a sequence or collection; any object that can be iterated through.
  - Can be mutable or immutable.
- 

## Built In Data Types

- **Lists** are the python equivalent of arrays.

```

1 empty_list = []
2 departments = ['HR','Development','Sales','Finance','IT','Customer Support']#
3 specials = list()# Test if a value is in a list.
4 print(1 in [1, 2, 3]) #> True
5 print(4 in [1, 2, 3]) #> False

```

- **Tuples** : Very similar to lists, but they are `immutable`

```

1 # Instantiated with parentheses
2 time_blocks = ('AM','PM')# Sometimes instantiated without
3 colors = 'red','blue','green'
4 numbers = 1, 2, 3# Tuple() built in can be used to convert other data into a tuple
5 tuple('abc') # returns ('a', 'b', 'c')
6 tuple([1,2,3]) # returns (1, 2, 3)

```

- Think of tuples as constant variables.
- **Ranges** : A list of numbers which can't be changed; often used with `for` loops.
- Declared using one to three parameters.
- **Start** : opt. default 0, first # in sequence.
- **Stop** : required next number past the last number in the sequence.
- **Step** : opt. default 1, difference between each number in the sequence.

```
1 range(5) # [0, 1, 2, 3, 4]
2 range(1,5) # [1, 2, 3, 4]
3 range(0, 25, 5) # [0, 5, 10, 15, 20]
4 range(0) # []for let (i = 0; i < 5; i++)
5 for let (i = 1; i < 5; i++)
6 for let (i = 0; i < 25; i+=5)
7 for let(i = 0; i = 0; i++)
```

- Keep in mind that `stop` is not included in the range.
- **Dictionaries** : Mappable collection where a hashable value is used as a key to ref. an object stored in the dictionary.
- Mutable.

```
1 a = {'one':1, 'two':2, 'three':3}
2 b = dict(one=1, two=2, three=3)
3 c = dict([('two', 2), ('one', 1), ('three', 3)])
```

*a, b, and c are all equal*

- Declared with curly braces of the built in `dict()`
- Benefit of dictionaries in Python is that it doesn't matter how it is defined, if the keys and values are the same the dictionaries are considered equal.
- Use the `in` operator to see if a key exists in a dictionary.

## Sets : Unordered collection of distinct objects; objects that need to be hashable.

- Always be unique, duplicate items are auto dropped from the set.
- **Common Uses:**
  - Removing Duplicates
  - Membership Testing
  - Mathematical Operators: Intersection, Union, Difference, Symmetric Difference.
  - Standard Set is mutable, Python has a immutable version called `frozenset` .
  - Sets created by putting comma seperated values inside braces:

```
1 school_bag = {'book', 'paper', 'pencil', 'pencil', 'book', 'book', 'book', 'eraser'}
2 print(school_bag) # Also can use set constructor to automatically put it into a
3 letters = set('abracadabra')
4 print(letters)
```

## Built-In Functions

### Functions using iterables

- **filter(function, iterable)** : creates new iterable of the same type which includes each item for which the function returns true.
- **map(function, iterable)** : creates new iterable of the same type which includes the result of calling the function on every item of the iterable.
- **sorted(iterable, key=None, reverse=False)** : creates a new sorted list from the items in the iterable.
- Output is always a `list`
- `key` : opt function which converts an item to a value to be compared.
- `reverse` : optional boolean.
- **enumerate(iterable, start=0)** : starts with a sequence and converts it to a series of tuples

```
1 quarters = ['First', 'Second', 'Third', 'Fourth']
2 print(enumerate(quarters))
3 print(enumerate(quarters, start=1)) # (0, 'First'), (1, 'Second'), (2, 'Third')
4 # (1, 'First'), (2, 'Second'), (3, 'Third'), (4, 'Fourth')
```

- **zip(\*iterables)** : creates a zip object filled with tuples that combine 1 to 1 the items in each provided iterable.

### Functions that analyze iterables

- **len(iterable)** : returns the count of the number of items.
- **max(\*args, key=None)** : returns the largest of two or more arguments.
- **max(iterable, key=None)** : returns the largest item in the iterable.
- `key` optional function which converts an item to a value to be compared.

- **min** works the same way as `max`
- **sum(iterable)** : used with a list of numbers to generate the total.
- There is a faster way to concatenate an array of strings into one string, so do not use sum for that.
- **any(iterable)** : returns True if any items in the iterable are true.
- **all(iterable)** : returns True if all items in the iterable are true.

## Working with dictionaries

- **dir(dictionary)** : returns the list of keys in the dictionary.

## Working with sets

- **Union** : The pipe `|` operator or `union(*sets)` function can be used to produce a new set which is a combination of all elements in the provided set.

```

1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a | b) # => {1, 2, 3, 4, 6}
```

- **Intersection** : The `&` operator can be used to produce a new set of only the elements that appear in all sets.

```

1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a & b) # => {2}
```

- **Difference** : The `-` operator can be used to produce a new set of only the elements that appear in the first set and NOT the others.
- **Symmetric Difference** : The `^` operator can be used to produce a new set of only the elements that appear in exactly one set and not in both.

```

1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a - b) # => {1, 3}
4 print(b - a) # => {4, 6}
```

```
5 print(a ^ b) # => {1, 3, 4, 6}
```

## For Statements

- In python, there is only one for loop.
- Always Includes:
  - The `for` keyword
  - A variable name
  - The `in` keyword
  - An iterable of some kind
  - A colon
- On the next line, an indented block of code called the `for` clause.
- You can use `break` and `continue` statements inside for loops as well.
- You can use the range function as the iterable for the `for` loop.

```
1 print('My name is')
2 for i in range(5):
3 print('Carlita Cinco (' + str(i) + ')')total = 0
4 for num in range(101):
5 total += num
6 print(total)
```

- Looping over a list in Python

```
1 for c in ['a', 'b', 'c']:
2 print(c)lst = [0, 1, 2, 3]
3 for i in lst:
4 print(i)
```

- Common technique is to use the `len()` on a pre-defined list with a for loop to iterate over the indices of the list.

```
1 supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
```

```
2 for i in range(len(supplies)):
3 print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

- You can loop and destructure at the same time.

```
1 l = [[1, 2], [3, 4], [5, 6]]
2 for a, b in l:
3 print(a, ', ', b)# Prints 1, 2
4 # Prints 3, 4
5 # Prints 5, 6
```

- You can use `values()` and `keys()` to loop over dictionaries.

```
1 spam = {'color': 'red', 'age': 42}
2 for v in spam.values():
3 print(v)# Prints red
4 # Prints 42for k in spam.keys():
5 print(k)# Prints color
6 # Prints age
```

- For loops can also iterate over both keys and values.

```
1 # Getting tuples
2 for i in spam.items():
3 print(i)# Prints ('color', 'red')
4 # Prints ('age', 42)
5 # Destructuring to values
6 for k, v in spam.items():
7 print('Key: ' + k + ' Value: ' + str(v))# Prints Key: age Value: 42
8 # Prints Key: color Value: red
```

- Looping over string

```
1 for c in "abcdefg":
2 print(c)
```

---

## More On Functions

- **Variable-length positional arguments : (\*args)**

```
1 def add(a, b, *args):
2 total = a + b;
3 for n in args:
4 total += n
5 return totaladd(1, 2) # Returns 3add(2, 3, 4, 5) # Returns 14
```

- **keyword arguments : (\*\*kwargs)**

```
1 def print_names_and_countries(greeting, **kwargs):
2 for k, v in kwargs.items():
3 print(greeting, k, "from", v)print_names_and_countries("Hi",
4 Monica="Sweden",
5 Charles="British Virgin Islands",
6 Carlo="Portugal")
7 # Prints
8 # Hi Monica from Sweden
9 # Hi Charles from British Virgin Islands
10 # Hi Carlo from Portugal
```

- When you order arguments within a function or function call, the args need to occur in a particular order:
- formal positional args.
- \*args
- keyword args with default values
- \*\*kwargs

```
1 def example(arg_1, arg_2, *args, **kwargs):
2 passdef example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blowfish", **kwargs)
3 pass
```

# Importing in Python

- Modules are similar to packages in Node.js
- Come in different types: Built-In, Third-Party, Custom.
- All loaded using `import` statements.

## Terms

- **module** : Python code in a separate file.
- **package** : Path to a directory that contains modules.
- **\*\*init.py\*\*** : Default file for a package.
- **submodule** : Another file in a module's folder.
- **function** : Function in a module.
- A module can be any file but it is usually created by placing a special file `__init__.py` into a folder.
- Try to avoid importing with wildcards in Python.
- Use multiple lines for clarity when importing.

```
1 from urllib.request import (
2 HTTPDefaultErrorHandler as ErrorHandler,
3 HTTPRedirectHandler as RedirectHandler,
4 Request,
5 pathname2url,
6 url2pathname,
7 urlopen,
8)
```

# Watching Out for Python 2

- Python 3 removed `<>` and only uses `!=`
- `format()` was introduced with P3
- All strings in P3 are unicode and encoded.
- `md5` was removed.
- `ConfigParser` was renamed to `configparser`
- `sets` were killed in favor of `set()` class.
- `print` was a statement in P2, but is a function in P3.

# Classes In Python

- Classes are a way of combining information and behavior.
- Classes are blueprints to make objects.

```
1 class AngryBird {
2 constructor() {
3 this.x = 0;
4 this.y = 0;
5 }
6 }
7 class AngryBird:
8 def __init__(self):
9 """
10 Construct a new AngryBird by setting its position to (0, 0).
11 """
12 self.x = 0
13 self.y = 0
```

- Both JS and PY use the `class` keyword to declare classes.
- `constructor == __init__`
- `this == self`

```
1 bird = AngryBird()
2 print(bird.x, bird.y) #> 0 0
3 class AngryBird:
4 def __init__(self):
5 """
6 Construct a new AngryBird by setting its position to (0, 0).
7 """
8 self.x = 0
9 self.y = 0 def move_up_by(self, delta):
10 self.y += delta
```

- Note how you do not need to define `self` it is already bound to the class.
- It is good practice to write a comment at the beginning of your class, describing the class.

## Dunder Methods

- Double Underscore Methods, special built in functions that PY uses in certain ways.

- i.e. `__init__()` lets you make sure all relevant attributes are set to their proper values when an object is created from the class.
- The `self` keyword refers to the current object that you are working with.
- Method is a function that is part of a class.

```

1 class AngryBird:
2 def __init__(self):
3 self.x = 0
4 self.y = 0 def move_up_by(self, delta):
5 self.y += delta
6 bird = AngryBird()
7 print(bird)
8 print(bird.y)
9 bird.move_up_by(5)
10 print(bird.y)

```

- *Use one leading underscore only for non-public methods and instance variables*

```

1 class AngryBird:
2 def __init__(self, x=0, y=0):
3 """
4 Construct a new AngryBird by setting its position to (0, 0).
5 """
6 self._x = x
7 self._y = y def move_up_by(self, delta):
8 self._y += delta def get_x(self):
9 return self._x def get_y(self):
10 return self._y

```

- *All instance variables should be considered non-public*
- **\*\*slots\*\*** : Dunder class variable used to reserve memory for the instance variables that you know will you will use.

```

1 class AngryBird:
2 __slots__ = ['_x', '_y'] def __init__(self, x=0, y=0):
3 """
4 Construct a new AngryBird by setting its position to (0, 0).
5 """
6 self._x = x
7 self._y = y def move_up_by(self, delta):

```

```
8 self._y += delta def get_x(self):
9 return self._x def get_y(self):
10 return self._y
```

- You can use `__repr__()` to override the behavior of printing out a class in a verbose manner.

```
1 class AngryBird:
2 __slots__ = ['_x', '_y'] def __init__(self, x=0, y=0):
3 """
4 Construct a new AngryBird by setting its position to (0, 0).
5 """
6 self._x = x
7 self._y = y def move_up_by(self, delta):
8 self._y += delta def get_x(self):
9 return self._x def get_y(self):
10 return self._y def __repr__(self):
11 return f"<AngryBird ({self._x}, {self._y})>"
```

## Properties for Classes

- Getters and Setters are used in object-oriented programming to add validation logic around getting and setting a value.

### Getters

```
bird = AngryBird()print(bird.get_x(), bird.get_y())
```

- Getting the x and y values of our class can get very cumbersome.
- **Decorators** : Allow us to change the way methods get invoked.
- Always start with the @ symbol.
- Can be applied to methods, classes, and parameters.
- Built in decorator named `property` that you can apply to a method to make it readable.

```

1 @property
2 def x(self):
3 return self._x @property
4 def y(self):
5 return self._y bird = AngryBird() print(bird.x, bird.y)

```

## Setters

```

1 class AngryBird:
2 def __init__(self, x=0, y=0):
3 """
4 Construct a new AngryBird by setting its position to (0, 0).
5 """
6 self._x = x
7 self._y = y def move_up_by(self, delta):
8 self._y += delta @property
9 def x(self):
10 return self._x @x.setter
11 def x(self, value):
12 if value < 0:
13 value = 0
14 self._x = value @property
15 def y(self):
16 return self._y @y.setter
17 def y(self, value):
18 if value < 0:
19 value = 0
20 self._y = value

```

## List Comprehensions

- List comprehensions are the equivalent of wrapped up filter map array methods while also allowing nested loops.
- `new_list = [expression for member in iterable]`
- **expression** : member itself, a call to a method, or any other valid expression that returns a value.
- **member** : object or value in the list or iterable.
- **iterable** : iterable.

```
new_list = [expression for member in iterable (if conditional)]
```

- Adding a conditional into a list comprehension.

```
1 sentence = 'Mary, Mary, quite contrary, how does your garden grow?'
2 def is_consonant(letter):
3 vowels = "aeiou"
4 return letter.isalpha() and letter.lower() not in vowels
5 # Prints ['M', 'r', 'y', 'M', 'r', 'y', 'q', 't', 'c',
6 # 'n', 't', 'r', 'r', 'y', 'h', 'w', 'd', 's', 'y',
7 # 'r', 'g', 'r', 'd', 'n', 'g', 'r', 'w']
```

## When to not use list comprehensions

- List comprehensions may make your code run more slowly or use more memory.
- You can use nest lists to create matrices.

```
1 matrix = [[i for i in range(5)] for _ in range(6)]print(matrix)
2 # Prints
3 # [
4 # [0, 1, 2, 3, 4],
5 # [0, 1, 2, 3, 4],
6 # [0, 1, 2, 3, 4],
7 # [0, 1, 2, 3, 4],
8 # [0, 1, 2, 3, 4],
9 # [0, 1, 2, 3, 4]
10 #]
```

## My Blog:

Web-Dev-HubMemoization, Tabulation, and Sorting Algorithms by Example Why is looking at runtime not a reliable method of...master-bgoonz-blog.netlify.app

## Python Data Structures & Algorithms Resources:

- The Framework for Learning Algorithms and intense problem solving exercises
- Algs4: Recommended book for Learning Algorithms and Data Structures
- An analysis of Dynamic Programming
- Dynamic Programming Q&A – What is Optimal Substructure
- The Framework for Backtracking Algorithm
- Binary Search in Detail: I wrote a Poem
- The Sliding Window Technique
- Difference Between Process and Thread in Linux
- Some Good Online Practice Platforms
- Dynamic Programming in Details
- Dynamic Programming Q&A – What is Optimal Substructure
- Classic DP: Longest Common Subsequence
- Classic DP: Edit Distance
- Classic DP: Super Egg
- Classic DP: Super Egg (Advanced Solution)
- The Strategies of Subsequence Problem
- Classic DP: Game Problems
- Greedy: Interval Scheduling
- KMP Algorithm In Detail
- A solution to all Buy Time to Buy and Sell Stock Problems
- A solution to all House Robber Problems
- 4 Keys Keyboard
- Regular Expression
- Longest Increasing Subsequence
- The Framework for Learning Algorithms and intense problem solving exercises
- Algs4: Recommended book for Learning Algorithms and Data Structures
- Binary Heap and Priority Queue
- LRU Cache Strategy in Detail
- Collections of Binary Search Operations
- Special Data Structure: Monotonic Stack
- Special Data Structure: Monotonic Stack
- Design Twitter
- Reverse Part of Linked List via Recursion
- Queue Implement Stack/Stack implement Queue
- My Way to Learn Algorithm
- The Framework of Backtracking Algorithm
- Binary Search in Detail
- Backtracking Solve Subset/Permutation/Combination
- Diving into the technical parts of Double Pointers

- Sliding Window Technique
  - The Core Concept of TwoSum Problems
  - Common Bit Manipulations
  - Breaking down a Complicated Problem: Implement a Calculator
  - Pancake Sorting Algorithm
  - Prefix Sum: Intro and Concept
  - String Multiplication
  - FloodFill Algorithm in Detail
  - Interval Scheduling: Interval Merging
  - Interval Scheduling: Intersections of Intervals
  - Russian Doll Envelopes Problem
  - A collection of counter-intuitive Probability Problems
  - Shuffle Algorithm
  - Recursion In Detail
  - How to Implement LRU Cache
  - How to Find Prime Number Efficiently
  - How to Calculate Minimum Edit Distance
  - How to use Binary Search
  - How to efficiently solve Trapping Rain Water Problem
  - How to Remove Duplicates From Sorted Array
  - How to Find Longest Palindromic Substring
  - How to Reverse Linked List in K Group
  - How to Check the Validation of Parenthesis
  - How to Find Missing Element
  - How to Find Duplicates and Missing Elements
  - How to Check Palindromic LinkedList
  - How to Pick Elements From an Infinite Arbitrary Sequence
  - How to Schedule Seats for Students
  - Union-Find Algorithm in Detail
  - Union-Find Application
  - Problems that can be solved in one line
  - Find Subsequence With Binary Search
  - Difference Between Process and Thread in Linux
  - You Must Know About Linux Shell
  - You Must Know About Cookie and Session
  - Cryptology Algorithm
  - Some Good Online Practice Platforms
-

## Algorithms:

- 100 days of algorithms
- Algorithms – Solved algorithms and data structures problems in many languages.
- Algorithms by Jeff Erickson (Code) (HN)
- Top algos/DS to learn
- Some neat algorithms
- Mathematical Proof of Algorithm Correctness and Efficiency (2019)
- Algorithm Visualizer – Interactive online platform that visualizes algorithms from code.
- Algorithms for Optimization book
- Collaborative book on algorithms (Code)
- Algorithms in C by Robert Sedgewick
- Algorithm Design Manual
- MIT Introduction to Algorithms course (2011)
- How to implement an algorithm from a scientific paper (2012)
- Quadsort – Stable non-recursive merge sort named quadsort.
- System design algorithms – Algorithms you should know before system design.
- Algorithms Design book
- Think Complexity
- All Algorithms implemented in Rust
- Solutions to Introduction to Algorithms book (Code)
- Maze Algorithms (2011) (HN)
- Algorithmic Design Paradigms book (Code)
- Words and buttons Online Blog (Code)
- Algorithms animated
- Cache Oblivious Algorithms (2020) (HN)
- You could have invented fractional cascading (2012)
- Guide to learning algorithms through LeetCode (Code) (HN)
- How hard is unshuffling a string?
- Optimization Algorithms on Matrix Manifolds
- Problem Solving with Algorithms and Data Structures (HN) (PDF)
- Algorithms implemented in Python
- Algorithms implemented in JavaScript
- Algorithms & Data Structures in Java
- Wolfsort – Stable adaptive hybrid radix / merge sort.
- Evolutionary Computation Bestiary – Bestiary of evolutionary, swarm and other metaphor-based algorithms.

- Elements of Programming book – Decomposing programs into a system of algorithmic components. (Review) (HN) (Lobsters)
  - Competitive Programming Algorithms
  - CPP/C – C/C++ algorithms/DS problems.
  - How to design an algorithm (2018)
  - CSE 373 – Introduction to Algorithms, by Steven Skiena (2020)
  - Computer Algorithms II course (2020)
  - Improving Binary Search by Guessing (2019)
  - The case for a learned sorting algorithm (2020) (HN)
  - Elementary Algorithms – Introduces elementary algorithms and data structures. Includes side-by-side comparisons of purely functional realization and their imperative counterpart.
  - Combinatorics Algorithms for Coding Interviews (2018)
  - Algorithms written in different programming languages (Web)
  - Solving the Sequence Alignment problem in Python (2020)
  - The Sound of Sorting – Visualization and “Audibilization” of Sorting Algorithms. (Web)
  - Miniselect: Practical and Generic Selection Algorithms (2020)
  - The Slowest Quicksort (2019)
  - Functional Algorithm Design (2020)
  - Algorithms To Live By – Book Notes
  - Numerical Algorithms (2015)
  - Using approximate nearest neighbor search in real world applications (2020)
  - In search of the fastest concurrent Union-Find algorithm (2019)
  - Computer Science 521 Advanced Algorithm Design
- 

## Data Structures:

- Data Structures and Algorithms implementation in Go
- Which algorithms/data structures should I “recognize” and know by name?
- Dictionary of Algorithms and Data Structures
- Phil’s Data Structure Zoo
- The Periodic Table of Data Structures (HN)
- Data Structure Visualizations (HN)
- Data structures to name-drop when you want to sound smart in an interview
- On lists, cache, algorithms, and microarchitecture (2019)
- Topics in Advanced Data Structures (2019) (HN)
- CS166 Advanced DS Course (2019)

- Advanced Data Structures (2017) (HN)
- Write a hash table in C
- Python Data Structures and Algorithms
- HAMTs from Scratch (2018)
- JavaScript Data Structures and Algorithms
- Implementing a Key-Value Store series
- Open Data Structures – Provide a high-quality open content data structures textbook that is both mathematically rigorous and provides complete implementations. (Code)
- A new analysis of the false positive rate of a Bloom filter (2009)
- Ideal Hash Trees
- RRB-Trees: Efficient Immutable Vectors
- Some data structures and algorithms written in OCaml
- Let's Invent B(+)-Trees (HN)
- Anna – Low-latency, cloud-native KVS.
- Persistent data structures thanks to recursive type aliases (2019)
- Log-Structured Merge-Trees (2020)
- Bloom Filters for the Perplexed (2017)
- Understanding Bloom Filters (2020)
- Dense vs. Sparse Indexes (2020)
- Data Structures and Algorithms Problems
- Data Structures & Algorithms I Actually Used Working at Tech Companies (2020) (Lobsters) (HN)
- Let's implement a Bloom Filter (2020) (HN)
- Data Structures Part 1: Bulk Data (2019) (Lobsters)
- Data Structures Explained
- Introduction to Cache-Oblivious Data Structures (2018)
- The Daily Coding newsletter – Master JavaScript and Data Structures.
- Lectures Note for Data Structures and Algorithms (2019)
- Mechanically Deriving Binary Tree Iterators with Continuation Defunctionalization (2020)
- Segment Tree data structure
- Structure of a binary state tree (2020)
- Introductory data structures and algorithms
- Applying Textbook Data Structures for Real Life Wins (2020) (HN)
- Michael Scott – Nonblocking data structures lectures (2020) – Nonblocking concurrent data structures are an increasingly valuable tool for shared-memory parallel programming.
- Scal – High-performance multicore-scalable data structures and benchmarks. (Web)
- Hyperbolic embedding implementations
- Morphisms of Computational Constructs – Visual catalogue + story of morphisms displayed across computational structures.

- What is key-value store? (build-your-own-x) (2020)
- Lesser Known but Useful Data Structures
- Using Bloom filters to efficiently synchronize hash graphs (2020)
- Bloom Filters by Example (Code)
- Binary Decision Diagrams (HN)
- 3 Steps to Designing Better Data Structures (2020)
- Sparse Matrices (2019) (HN)
- Algorithms & Data Structures in C++
- Fancy Tree Traversals (2019)
- The Robson Tree Traversal (2019)
- Data structures and program structures
- cdb – Fast, reliable, simple package for creating and reading constant databases.
- PGM-index – Learned indexes that match B-tree performance with 83x less space. (HN)  
(Code)
- Structural and pure attributes
- Cache-Tries: O(1) Concurrent Lock-Free Hash Tries (2018)

=====

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf



Interview-Questions-Solved.md

<https://gist.github.com/bgoonz/4006e76c6a67b9023e0839ea863ab14f>

First things first, the repo with all the exercises of this lecture is right here:

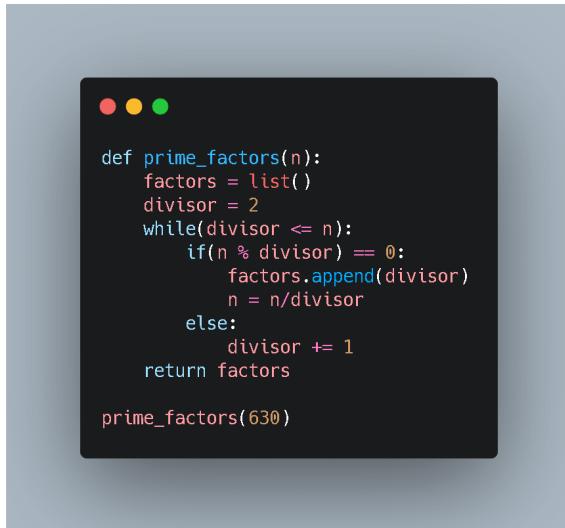
||||| [https://github.com/hugoestradas/Python\\_Basics](https://github.com/hugoestradas/Python_Basics)

Let's begin!

1) Find prime factors.

For the very basics, let's start with something unusual: Public Key Encryption. This technique relies on certain really large numbers being computationally hard to factor to keep data secure. In this first exercise I'll factor some numbers that are easy to deal with; the goal is to create a Python function to find all prime factors, I'll do it by taking an integer value as input and the return or output will be a list of prime factors.

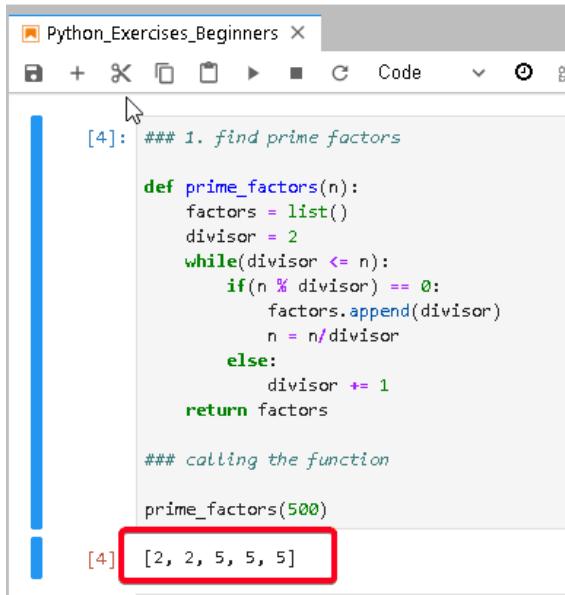
In this solution I decided to search for factors by dividing the given sequentially larger values (starting from 2) to see which one divide evenly into it, without leaving a remainder behind:



```
def prime_factors(n):
 factors = []
 divisor = 2
 while(divisor <= n):
 if(n % divisor) == 0:
 factors.append(divisor)
 n = n/divisor
 else:
 divisor += 1
 return factors

prime_factors(630)
```

As you can see I'm calling the function with the 500 number, so it will begin with 2 as the original divisor, then it'll go on keep dividing until the remainder is no longer an even number, in this case resulting in the result of 2, 2, 5, and finally 5:



```
[4]: ### 1. find prime factors

def prime_factors(n):
 factors = []
 divisor = 2
 while(divisor <= n):
 if(n % divisor) == 0:
 factors.append(divisor)
 n = n/divisor
 else:
 divisor += 1
 return factors

calling the function

prime_factors(500)
```

[4] [2, 2, 5, 5]

## 2) Identifying Palindromes.

This a very usual programming and software engineering exercise, maybe you already did it on colleague, school or watching another tutorial, it's a very cool puzzle to solve because involves pattern recognition, logic and of course coding.

In case it's your first time dealing with palindromes, a palindrome is a word or text that reads exactly the same, either forwards or backwards.

Again, I'll write a function to detect palindromes, where my input will be the string I'm checking and the result or output is going to be a boolean value (false/true):



```
import re

def palindromes(string):
 forwards = ''.join(re.findall(r'[a-z]+', string.lower()))
 backwards = forwards[::-1]
 return forwards == backwards

calling the function

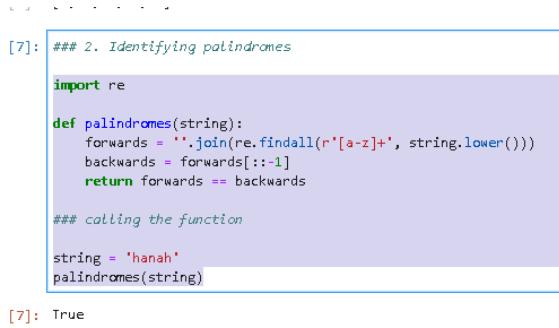
string = 'hanah'
palindromes(string)
```

Going line by line, first I'm importing the "re" library, which contains regular expressions to extract letters from an input string, then I'm defining a "palindrome" function that receives a "string" parameter.

Then I use the lower operator in the input string to convert all of the letters to lowercase, then I pass the result to the regular expression "findall" function with a pattern that will search for combinations of one or more letters. That will produce a list with all of the matched sub-strings that I merged together into a single string using the "join" function.

Then I slice the entire string, with the stride set to negative one, meaning I'll get a copy of the original string in reverse order.

Finally, I'm comparing both strings and return it:



```
[7]: ### 2. Identifying palindromes

import re

def palindromes(string):
 forwards = ''.join(re.findall(r'[a-z]+', string.lower()))
 backwards = forwards[::-1]
 return forwards == backwards

calling the function

string = 'hanah'
palindromes(string)
```

[7]: True

```

import re

def palindromes(string):
 forwards = ''.join(re.findall(r'[a-z]+', string.lower()))
 backwards = forwards[::-1]
 return forwards == backwards

calling the function

string = 'hugo'
palindromes(string)

[1]: False

```

### 3) Sort a string.

Another common task in programming is sorting things.

The goal is to create a Python function that sorts the words within a given string.

The input will be a list of words separated by spaces, and the result or output will be the same string of words sorted alphabetically:

```

● ● ●

def sorted(strings):
 words = strings.split()
 words = [w.lower() + w for w in words]
 words.sort()
 words = [w[:len(w)//2:] for w in words]
 return ' '.join(words)

calling the function

sorted("this is a list of words not sorted alphabetically")

```

My "sorted" function starts with the "split" method, which breaks apart the input string at each of the spaces and gives me a list of the individual word.

Then, to ignore the capitalization (if there is any) in the loop I convert each word within the list into lower case, to later on sort the entire list:

```

L = J = 0

[11]: ### 3. Sort a string

def sorted(strings):
 words = strings.split()
 words = [w.lower() + w for w in words]
 words.sort()
 words = [w[:len(w)//2:] for w in words]
 return ' '.join(words)

calling the function

sorted("this is a list of words not sorted alphabetically")

[11]: 'a alphabetically is list not of sorted this words'

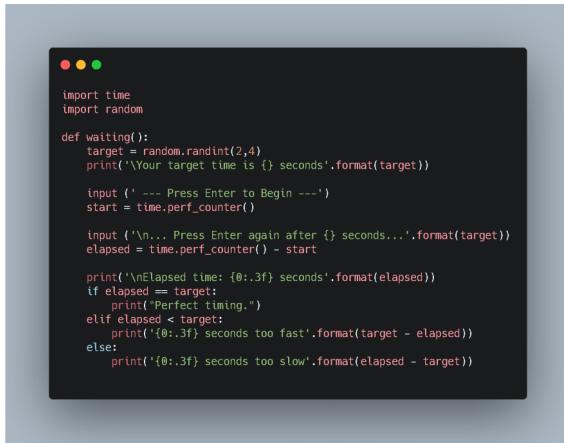
[]:

```

### 4) The waiting game.

For this exercise I'll write a Python function, which is when invoked it'll print a message to wait a random amount of time.

The user press enters, then the timer starts. The user's goal is to wait the specified number of seconds in the message, and then press enter again.



A screenshot of a terminal window on a Mac OS X desktop. The window title is 'Python 3.6.4'. The code inside the terminal is:

```
import time
import random

def waiting():
 target = random.randint(2,4)
 print('Your target time is {} seconds'.format(target))

 input ('--- Press Enter to Begin ---')
 start = time.perf_counter()

 input ('\n... Press Enter again after {} seconds...'.format(target))
 elapsed = time.perf_counter() - start

 print('\nElapsed time: {:.3f} seconds'.format(elapsed))
 if elapsed == target:
 print("Perfect timing.")
 elif elapsed < target:
 print('{:.3f} seconds too fast'.format(target - elapsed))
 else:
 print('{:.3f} seconds too slow'.format(elapsed - target))
```

For this exercise I used to modules, "time" module to measure the amount of time, and the "random" module to generate a random number of seconds.

The input function prompts the user to press enter to begin and then blocks the execution until the user hits enter again.



A screenshot of a terminal window showing the execution of the script. The output is:

```
[15]: #### 4. Waiting Game
import time
import random

def waiting():
 target = random.randint(2,4)
 print('Your target time is {} seconds'.format(target))

 input ('--- Press Enter to Begin ---')
 start = time.perf_counter()

 input ('\n... Press Enter again after {} seconds...'.format(target))
 elapsed = time.perf_counter() - start

 print('\nElapsed time: {:.3f} seconds'.format(elapsed))
 if elapsed == target:
 print("Perfect timing.")
 elif elapsed < target:
 print('{:.3f} seconds too fast'.format(target - elapsed))
 else:
 print('{:.3f} seconds too slow'.format(elapsed - target))

playing
waiting()
Your target time is 2 seconds
--- Press Enter to Begin ---

... Press Enter again after 2 seconds...
Elapsed time: 2.869 seconds
0.869 seconds too slow
```

## 5) Generate a new password.

For this final example, I'll implement a function based on the "Diceware" method, which is a method for creating passphrases and passwords using the numbers of an ordinary dice as hardware random number generator. It involves a list of over 7000 different words.

Instead of rolling a physical dice, I'll write a Python function that simulates this behavior.

The input will be a number of words in a passphrase and the output or result will be a string of random words, separated by spaces.

For this one, I could've used the "random" module, but instead I went for the "secret" module, since the random module is not recommended when dealing with cryptographic procedures:



```
import secrets
def passphrase(n):
 with open('diceware.wordlist.asc', 'r') as file:
 lines = file.readlines()[2:778]
 word_list = [line.split()[1] for line in lines]

 words = [secrets.choice(word_list) for i in range(n)]
 return ' '.join(words)
```

My function begins by getting the number of words, then opening the "diceware.wordlist.asc" file with a context manager and then uses "readlines" function to get a list with each of the lines within the file.

The top of the file diceware that I used has two extra lines before the word list actually begins, and at the bottom there are also several extra lines for a PGP signature:

```
1 |----BEGIN PGP SIGNED MESSAGE-----
2 |
3 11111--a
4 11112--a&p
5 11113--a*s
6 11114--aa
7 11115--aaa
8 11116--aaaa
9 11121--aaron
0 11122--ab
1 11123--aba
2 11124--ababa
3 11125--aback
4 11126--abase
5 11131--abash
6 11132--abate
7 11133--abbas
8 11134--abbe
9 11135--abbey
0 11136--ahhot
```

```

Python_Exercises_Beginners > diceware.wordlist.asc >
7762 66642--$$
7763 66643--%
7764 66644--%%
7765 66645--&
7766 66646--(
7767 66651--()
7768 66652--)
7769 66653--*
7770 66654--**
7771 66655--+
7772 66656---
7773 66661--:
7774 66662--;
7775 66663--=
7776 66664--?
7777 66665--??
7778 66666--@
7779
7780 -----BEGIN PGP SIGNATURE-----
7781 Version: PGP for Personal Privacy 5.0
7782 Charset: noconv
7783
7784 iQCVAwUBOn7XUmtrC2sHYShAQHp44QAh5x14GkCvdpz1RyXkywa/nBlmVNrcect
7785 i/8z4jvFsBOJQgzRC/BdwDuFv2NVPbEjE33e8YXcOP6dnyCqzF0nmKpqNchNPHS3
7786 QICgAofis9azx1/0Zr4fxzl3ewRxldyW8TY9Vj6uayNAqy+myUDC5FZFSX3kOho
7787 bgR/yFB40fA=
7788 =c65y
7789 -----END PGP SIGNATURE-----
7790

```

So I indexed out the 7K (7776) lines from the middle of the file that I actually care about. Remembering that each of these lines contain both a five-digit number and the corresponding word, I used the split method to break them apart, and then build the list containing just the words.

Then I used the "secrets.choice" function within another list comprehension to build a list with the desired number of random words.

And finally, I used the join method to combine the random words into a single string with spaces between them:

```

[19]: ## 5. Generating a new password

import secrets
def passphrase(n):
 with open('diceware.wordlist.asc', 'r') as file:
 lines = file.readlines()[2:7778]
 word_list = [line.split()[1] for line in lines]

 words = [secrets.choice(word_list) for i in range(n)]
 return ' '.join(words)

calling the function
passphrase(5)

```

[19]: 'wg lisle moran anise balmy'

[ 1: ]

# index

```
1 import math
2
3
4 def say_hi(name):
5 """<---- Multi-Line Comments and Docstrings
6 This is where you put your content for help() to inform the user
7 about what your function does and how to use it
8 """
9 print(f"Hello {name}!")
10
11
12 print(say_hi("Bryan")) # Should get the print inside the function, then None
13 # Boolean Values
14 # Work the same as in JS, except they are title case: True and False
15 a = True
16 b = False
17 # Logical Operators
18 # != not, || = or, && = and
19 print(True and True)
20 print(True and not True)
21 print(True or True)
22 # Truthiness - Everything is True except...
23 # False - None, False, '', [], (), set(), range(0)
24 # Number Values
25 # Integers are numbers without a floating decimal point
26 print(type(3)) # type returns the type of whatever argument you pass in
27 # Floating Point values are numbers with a floating decimal point
28 print(type(3.5))
29 # Type Casting
30 # You can convert between ints and floats (along with other types...)
31 print(float(3)) # If you convert a float to an int, it will truncate the decimal
32 print(int(4.5))
33 print(type(str(3)))
34 # Python does not automatically convert types like JS
35 # print(17.0 + ' heyooo ' + 17) # TypeError
36 # Arithmetic Operators
37 # ** - exponent (comparable to Math.pow(num, pow))
38 # // - integer division
39 # There is no ++ or -- in Python
40 # String Values
41 # We can use single quotes, double quotes, or f' ' for string formats
42 # We can use triple single quotes for multiline strings
43 print(
44 """This here's a story
45 All about how
```

```
46 My life got twist
47 Turned upside down
48 """
49)
50 # Three double quotes can also be used, but we typically reserve these for
51 # multi-line comments and function docstrings (refer to lines 6-9)(Nice :D)
52 # We use len() to get the length of something
53 print(len("Bryan G")) # 7 characters
54 print(len(["hey", "ho", "hey", "hey", "ho"])) # 5 list items
55 print(len({1, 2, 3, 4, 5, 6, 7, 9})) # 8 set items
56 # We can index into strings, list, etc..self.
57 name = "Bryan"
58 for i in range(len(name)):
59 print(name[i]) # B, r, y, a, n
60 # We can index starting from the end as well, with negatives
61 occupation = "Full Stack Software Engineer"
62 print(occupation[-3]) # e
63 # We can also get ranges in the index with the [start:stop:step] syntax
64 print(occupation[0:4:1]) # step and stop are optional, stop is exclusive
65 print(occupation[::-4]) # beginning to end, every 4th letter
66 print(occupation[4:14:2]) # Let's get weird with it!
67 # NOTE: Indexing out of range will give you an IndexError
68 # We can also get the index og things with the .index() method, similar to in
69 print(occupation.index("Stack"))
70 print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
71 # We can count how many times a substring/item appears in something as well
72 print(occupation.count("S"))
73 print(
74 """Now this here's a story all about how
75 My life got twist turned upside down
76 I forget the rest but the the potato
77 smells like the potato""".count(
78 "the"
79)
80)
81 # We concatenate the same as Javascript, but we can also multiply strings
82 print("dog " + "show")
83 print("ha" * 10)
84 # We can use format for a multitude of things, from spaces to decimal places
85 first_name = "Bryan"
86 last_name = "Guner"
87 print("Your name is {0} {1}".format(first_name, last_name))
88 # Useful String Methods
89 print("Hello".upper()) # HELLO
90 print("Hello".lower()) # hello
91 print("HELLO".islower()) # False
92 print("HELLO".isupper()) # True
93 print("Hello".startswith("he")) # False
94 print("Hello".endswith("lo")) # True
95 print("Hello There".split()) # [Hello, There]
96 print("hello1".isalpha()) # False, must consist only of letters
97 print("hello1".isalnum()) # True, must consist of only letters and numbers
98 print("3215235123".isdecimal()) # True, must be all numbers
```

```

99 # True, must consist of only spaces/tabs/newlines
100 print("\n ".isspace())
101 # False, index 0 must be upper case and the rest lower
102 print("Bryan Guner".istitle())
103 print("Michael Lee".istitle()) # True!
104 # Duck Typing - If it walks like a duck, and talks like a duck, it must be a duck
105 # Assignment - All like JS, but there are no special keywords like let or const
106 a = 3
107 b = a
108 c = "heyoo"
109 b = ["reassignment", "is", "fine", "G!"]
110 # Comparison Operators - Python uses the same equality operators as JS, but no ==
111 # < - Less than
112 # > - Greater than
113 # <= - Less than or Equal
114 # >= - Greater than or Equal
115 # == - Equal to
116 # != - Not equal to
117 # is - Refers to exact same memory location
118 # not - !
119 # Precedence - Negative Signs(not) are applied first(part of each number)
120 # - Multiplication and Division(and) happen next
121 # - Addition and Subtraction(or) are the last step
122 # NOTE: Be careful when using not along with ==
123 print(not a == b) # True
124 # print(a == not b) # Syntax Error
125 print(a == (not b)) # This fixes it. Answer: False
126 # Python does short-circuit evaluation
127 # Assignment Operators - Mostly the same as JS except Python has **= and //=
128 # Flow Control Statements - if, while, for
129 # Note: Python smushes 'else if' into 'elif'!
130 if 10 < 1:
131 print("We don't get here")
132 elif 10 < 5:
133 print("Nor here...")
134 else:
135 print("Hey there!")
136 # Looping over a string
137 for c in "abcdefghijklmnopqrstuvwxyz":
138 print(c)
139 # Looping over a range
140 for i in range(5):
141 print(i + 1)
142 # Looping over a list
143 lst = [1, 2, 3, 4]
144 for i in lst:
145 print(i)
146 # Looping over a dictionary
147 spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
148 for v in spam.values():
149 print(v)
150 # Loop over a list of tuples and destructuring the values
151 # Assuming spam.items returns a list of tuples each containing two items (k, v)

```

```
152 for k, v in spam.items():
153 print(f"{k}: {v}")
154 # While loops as long as the condition is True
155 # - Exit loop early with break
156 # - Exit iteration early with continue
157 spam = 0
158 while True:
159 print("Sike That's the wrong Numba")
160 spam += 1
161 if spam < 5:
162 continue
163 break
164
165 # Functions - use def keyword to define a function in Python
166
167
168 def printCopyright():
169 print("Copyright 2021, Bgoonz")
170
171
172 # Lambdas are one liners! (Should be at least, you can use parenthesis to dis-
173 def avg(num1, num2):
174 return print(num1 + num2)
175
176
177 avg(1, 2)
178 # Calling it with keyword arguments, order does not matter
179 avg(num2=20, num1=1252)
180 printCopyright()
181 # We can give parameters default arguments like JS
182
183
184 def greeting(name, saying="Hello"):
185 print(saying, name)
186
187
188 greeting("Mike") # Hello Mike
189 greeting("Bryan", saying="Hello there...")
190 # A common gotcha is using a mutable object for a default parameter
191 # All invocations of the function reference the same mutable object
192
193
194 def append_item(item_name, item_list=[]): # Will it obey and give us a new li-
195 item_list.append(item_name)
196 return item_list
197
198
199 # Uses same item list unless otherwise stated which is counterintuitive
200 print(append_item("notebook"))
201 print(append_item("notebook"))
202 print(append_item("notebook", []))
203 # Errors - Unlike JS, if we pass the incorrect amount of arguments to a functi-
204 # it will throw an error
```

```

205 # avg(1) # TypeError
206 # avg(1, 2, 2) # TypeError
207 # ----- DAY 2 -----
208 # Functions - * to get rest of position arguments as tuple
209 # - ** to get rest of keyword arguments as a dictionary
210 # Variable Length positional arguments
211
212
213 def add(a, b, *args):
214 # args is a tuple of the rest of the arguments
215 total = a + b
216 for n in args:
217 total += n
218 return total
219
220
221 print(add(1, 2)) # args is None, returns 3
222 print(add(1, 2, 3, 4, 5, 6)) # args is (3, 4, 5, 6), returns 21
223 # Variable Length Keyword Arguments
224
225
226 def print_names_and_countries(greeting, **kwargs):
227 # kwargs is a dictionary of the rest of the keyword arguments
228 for k, v in kwargs.items():
229 print(greeting, k, "from", v)
230
231
232 print_names_and_countries(
233 "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
234)
235 # We can combine all of these together
236
237
238 def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
239 pass
240
241
242 # Lists are mutable arrays
243 empty_list = []
244 roomates = ["Beau", "Delynn"]
245 # List built-in function makes a list too
246 specials = list()
247 # We can use 'in' to test if something is in the list, like 'includes' in JS
248 print(1 in [1, 2, 4]) # True
249 print(2 in [1, 3, 5]) # False
250 # Dictionaries - Similar to JS POJO's or Map, containing key value pairs
251 a = {"one": 1, "two": 2, "three": 3}
252 b = dict(one=1, two=2, three=3)
253 # Can use 'in' on dictionaries too (for keys)
254 print("one" in a) # True
255 print(3 in b) # False
256 # Sets - Just like JS, unordered collection of distinct objects
257 bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}

```

```

258 # bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
259 school_bag = set(
260 ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"]
261)
262 print(school_bag)
263 print(bedroom)
264 # We can use 'in' on sets as well
265 print(1 in {1, 2, 3}) # True
266 print(4 in {1, 3, 5}) # False
267 # Tuples are immutable lists of items
268 time_blocks = ("AM", "PM")
269 colors = "red", "green", "blue" # Parenthesis not needed but encouraged
270 # The tuple built-in function can be used to convert things to tuples
271 print(tuple("abc"))
272 print(tuple([1, 2, 3]))
273 # 'in' may be used on tuples as well
274 print(1 in (1, 2, 3)) # True
275 print(5 in (1, 4, 3)) # False
276 # Ranges are immutable lists of numbers, often used with for loops
277 # - start - default: 0, first number in sequence
278 # - stop - required, next number past last number in sequence
279 # - step - default: 1, difference between each number in sequence
280 range1 = range(5) # [0,1,2,3,4]
281 range2 = range(1, 5) # [1,2,3,4]
282 range3 = range(0, 25, 5) # [0,5,10,15,20]
283 range4 = range(0) # []
284 for i in range1:
285 print(i)
286 # Built-in functions:
287 # Filter
288
289
290 def isOdd(num):
291 return num % 2 == 1
292
293
294 filtered = filter(isOdd, [1, 2, 3, 4])
295 print(list(filtered))
296 for num in filtered:
297 print(f"first way: {num}")
298 print("--" * 20)
299 [print(f"list comprehension: {i}") for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
300
301 # Map
302
303
304 def toUpper(str):
305 return str.upper()
306
307
308 upperCased = map(toUpper, ["a", "b", "c", "d"])
309 print(list(upperCased))
310 # Sorted

```

```

311 sorted_items = sorted(["john", "tom", "sonny", "Mike"])
312 print(list(sorted_items)) # Notice uppercase comes before lowercase
313 # Using a key function to control the sorting and make it case insensitive
314 sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
315 print(sorted_items)
316 # You can also reverse the sort
317 sorted_items = sorted(["john", "tom", "sonny", "Mike"],
318 key=str.lower, reverse=True)
319 print(sorted_items)
320 # Enumerate creates a tuple with an index for what you're enumerating
321 quarters = ["First", "Second", "Third", "Fourth"]
322 print(list(enumerate(quarters)))
323 print(list(enumerate(quarters, start=1)))
324 # Zip takes list and combines them as key value pairs, or really however you want
325 keys = ("Name", "Email")
326 values = ("Buster", "cheetoh@johhnydepp.com")
327 zipped = zip(keys, values)
328 print(list(zipped))
329 # You can zip more than 2
330 x_coords = [0, 1, 2, 3, 4]
331 y_coords = [4, 6, 10, 9, 10]
332 z_coords = [20, 10, 5, 9, 1]
333 coords = zip(x_coords, y_coords, z_coords)
334 print(list(coords))
335 # Len reports the length of strings along with list and any other object data type
336 # doing this to save myself some typing
337
338
339 def print_len(item):
340 return print(len(item))
341
342
343 print_len("Mike")
344 print_len([1, 5, 2, 10, 3, 10])
345 print_len({1, 5, 10, 9, 10}) # 4 because there is a duplicate here (10)
346 print_len((1, 4, 10, 9, 20))
347 # Max will return the max number in a given scenario
348 print(max(1, 2, 35, 1012, 1))
349 # Min
350 print(min(1, 5, 2, 10))
351 print(min([1, 4, 7, 10]))
352 # Sum
353 print(sum([1, 2, 4]))
354 # Any
355 print(any([True, False, False]))
356 print(any([False, False, False]))
357 # All
358 print(all([True, True, False]))
359 print(all([True, True, True]))
360 # Dir returns all the attributes of an object including its methods and dunder
361 user = {"Name": "Bob", "Email": "bob@bob.com"}
362 print(dir(user))
363 # Importing packages and modules

```

```
364 # - Module - A Python code in a file or directory
365 # - Package - A module which is a directory containing an __init__.py file
366 # - Submodule - A module which is contained within a package
367 # - Name - An exported function, class, or variable in a module
368 # Unlike JS, modules export ALL names contained within them without any special
369 # Assuming we have the following package with four submodules
370 # math
371 # | __init__.py
372 # | addition.py
373 # | subtraction.py
374 # | multiplication.py
375 # | division.py
376 # If we peek into the addition.py file we see there's an add function
377 # addition.py
378 # We can import 'add' from other places because it's a 'name' and is automatic
379
380
381 # def add(num1, num2):
382 # return num1 + num2
383
384
385 # Notice the . syntax because this package can import it's own submodules.
386 # Our __init__.py has the following files
387 # This imports the 'add' function
388 # And now it's also re-exported in here as well
389 # from .addition import add
390 # These import and re-export the rest of the functions from the submodule
391 # from .subtraction import subtract
392 # from .division import divide
393 # from .multiplication import multiply
394 # So if we have a script.py and want to import add, we could do it many ways
395 # This will load and execute the 'math/__init__.py' file and give
396 # us an object with the exported names in 'math/__init__.py'
397 # print(math.add(1,2))
398 # This imports JUST the add from 'math/__init__.py'
399 # from math import add
400 # print(add(1, 2))
401 # This skips importing from 'math/__init__.py' (although it still runs)
402 # and imports directly from the addition.py file
403 # from math.addition import add
404 # This imports all the functions individually from 'math/__init__.py'
405 # from math import add, subtract, multiply, divide
406 # print(add(1, 2))
407 # print(subtract(2, 1))
408 # This imports 'add' renames it to 'add_some_numbers'
409 # from math import add as add_some_numbers
410 # ----- DAY 3 -----
411 # Classes, Methods, and Properties
412
413
414 class AngryBird:
415 # Slots optimize property access and memory usage and prevent you
416 # from arbitrarily assigning new properties to the instance
```

```

417 __slots__ = ["_x", "_y"]
418 # Constructor
419
420 def __init__(self, x=0, y=0):
421 # Doc String
422 """
423 Construct a new AngryBird by setting it's position to (0, 0)
424 """
425 # Instance Variables
426 self._x = x
427 self._y = y
428
429 # Instance Method
430
431 def move_up_by(self, delta):
432 self._y += delta
433
434 # Getter
435
436 @property
437 def x(self):
438 return self._x
439
440 # Setter
441
442 @x.setter
443 def x(self, value):
444 if value < 0:
445 value = 0
446 self._x = value
447
448 @property
449 def y(self):
450 return self._y
451
452 @y.setter
453 def y(self, value):
454 self._y = value
455
456 # Dunder Repr... called by 'print'
457
458 def __repr__(self):
459 return f"<AngryBird ({self._x}, {self._y})>"
460
461
462 # JS to Python Classes cheat table
463 # JS Python
464 # constructor() def __init__(self):
465 # super() super().__init__()
466 # this.property self.property
467 # this.method self.method()
468 # method(arg1, arg2){} def method(self, arg1, ...)
469 # get someProperty(){} @property

```

```
470 # set someProperty(){} @someProperty.setter
471 # List Comprehensions are a way to transform a list from one format to another
472 # - Pythonic Alternative to using map or filter
473 # - Syntax of a list comprehension
474 # - new_list = [value loop condition]
475 # Using a for loop
476 squares = []
477 for i in range(10):
478 squares.append(i ** 2)
479 print(squares)
480 # value = i ** 2
481 # loop = for i in range(10)
482 squares = [i ** 2 for i in range(10)]
483 print(list(squares))
484 sentence = "the rocket came back from mars"
485 vowels = [character for character in sentence if character in "aeiou"]
486 print(vowels)
487 # You can also use them on dictionaries. We can use the items() method
488 # for the dictionary to loop through it getting the keys and values out at once
489 person = {"name": "Corina", "age": 32, "height": 1.4}
490 # This loops through and capitalizes the first letter of all keys
491 newPerson = {key.title(): value for key, value in person.items()}
492 print(list(newPerson.items()))
```

# My Python DS Websites

<<<<< HEAD

Python Data Structures

Number Base Converter

Awesome Search

The Algos

Ds-Pac

Docs



ds-algo (forked) - CodeSandbox

[https://codesandbox.io/s/ds-algo-forked-lfujh?  
file=/index.html&resolutionWidth=1400  
&resolutionHeight=1259](https://codesandbox.io/s/ds-algo-forked-lfujh?file=/index.html&resolutionWidth=1400&resolutionHeight=1259)

hi

=====



e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf



Docs

<https://thealgorithms.netlify.app/#>



DS-Algo-Codebase

[https://bgoonz-branch-the-  
algos.vercel.app/](https://bgoonz-branch-the-algos.vercel.app/)



DS-Algo-Codebase

[https://bgoonz-branch-the-  
algos.vercel.app/](https://bgoonz-branch-the-algos.vercel.app/)

# Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

## Built-in Functions

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

`abs (x)`

Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned. `all (iterable)`

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
1 def all(iterable):
2 for element in iterable:
3 if not element:
4 return False
5 return True
```

`any (iterable)`

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`.

Equivalent to:

```
1 def any(iterable):
2 for element in iterable:
3 if element:
4 return True
5 return False
```

`ascii (object)`

As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2. `bin (x)`

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:>>>

```
1 >>> bin(3)
2 '0b11'
3 >>> bin(-10)
4 '-0b1010'
```

If prefix “0b” is desired or not, you can use either of the following ways.>>>

```
1 >>> format(14, '#b'), format(14, 'b')
2 ('0b1110', '1110')
3 >>> f'{14:#b}', f'{14:b}'
4 ('0b1110', '1110')
```

See also `format()` for more information.

`class bool ([x])`

Return a Boolean value, i.e. one of `True` or `False`. `x` is converted using the standard truth testing procedure. If `x` is false or omitted, this returns `False`; otherwise it returns `True`. The `bool` class is a subclass of `int` (see Numeric Types – `int`, `float`, `complex`). It cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Values).

Changed in version 3.7: `x` is now a positional-only parameter. `breakpoint (*args, **kws)`

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import `pdb` or type as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and `breakpoint()` will automatically call that, allowing you to drop into the debugger of choice.

Raises an auditing event `builtins.breakpoint` with argument `breakpointhook`.

New in version 3.7.`class bytearray ([source[, encoding[, errors]])`

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers in the range  $0 \leq x < 256$ . It has most of the usual methods of mutable sequences, described in Mutable Sequence Types, as well as most methods that the `bytes` type has, see Bytes and Bytearray Operations.

The optional `source` parameter can be used to initialize the array in a few different ways:

- If it is a `string`, you must also give the `encoding` (and optionally, `errors`) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an `integer`, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array.

- If it is an *iterable*, it must be an iterable of integers in the range `0 <= x < 256`, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also [Binary Sequence Types – bytes, bytearray, memoryview and Bytearray Objects](#).  
`bytes ([source], encoding[, errors])`

Return a new “bytes” object, which is an immutable sequence of integers in the range `0 <= x < 256`. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for `bytearray()`.

Bytes objects can also be created with literals, see [String and Bytes literals](#).

See also [Binary Sequence Types – bytes, bytearray, memoryview, Bytes Objects, and Bytes and Bytearray Operations](#). `callable (object)`

Return `True` if the *object* argument appears callable, `False` if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2. `chr (i)`

Return the string representing a character whose Unicode code point is the integer *i*. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. This is the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16).

`ValueError` will be raised if *i* is outside that range. `@classmethod`

Transform a method into a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
1 class C:
2 @classmethod
3 def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function decorator – see [Function definitions](#) for details.

A class method can be called either on the class (such as `c.f()`) or on an instance (such as `c().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section. For more information on class methods, see [The standard type hierarchy](#).

Changed in version 3.9: Class methods can now wrap other descriptors such as `property()`.  
`compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)`

Compile the `source` into a code or AST object. Code objects can be executed by `exec()` or `eval()`. `source` can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file (`'<string>'` is commonly used).

The `mode` argument specifies what kind of code must be compiled; it can be `'exec'` if `source` consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments `flags` and `dont_inherit` control which compiler options should be activated and which future features should be allowed. If neither is present (or both are zero) the code is compiled with the same flags that affect the code that is calling `compile()`. If the `flags` argument is given and `dont_inherit` is not (or is zero) then the compiler options and the future statements specified by the `flags` argument are used in addition to those that would be used anyway. If `dont_inherit` is a non-zero integer then the `flags` argument is it – the flags (future features and compiler options) in the surrounding code are ignored.

Compiler options and future statements are specified by bits which can be bitwise ORed together to specify multiple options. The bitfield required to specify a given future feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module. Compiler flags can be found in `ast` module, with `PyCF_` prefix.

The argument `optimize` specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an auditing event `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

## Note

When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

## Warning

It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also input in `'exec'` mode does not have to end in a newline anymore. Added the `optimize` parameter.

Changed in version 3.5: Previously, `TypeError` was raised when null bytes were encountered in `source`.

New in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in flags to enable support for top-level `await`, `async for`, and `async with .class complex ([real[, imag]])`

Return a complex number with the value `real + imag*1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If `imag` is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

## Note

When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in Numeric Types – `int`, `float`, `complex`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined. `delattr(object, name)`

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x foobar`.

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and Mapping Types – `dict` for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module. `dir([object])`

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:>>>

```
1 >>> import struct
2 >>> dir() # show the names in the module namespace
3 ['__builtins__', '__name__', 'struct']
4 >>> dir(struct) # show the names in the struct module
5 ['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
6 '__initializing__', '__loader__', '__name__', '__package__',
7 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
8 'unpack', 'unpack_from']
9 >>> class Shape:
10 ... def __dir__(self):
11 ... return ['area', 'perimeter', 'location']
12 >>> s = Shape()
13 >>> dir(s)
14 ['area', 'location', 'perimeter']
```

## Note

Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class. `divmod(a, b)`

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where `q` is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to `a`, if `a % b` is non-zero it has the same sign as `b`, and `0 <= abs(a % b) < abs(b)`. `enumerate(iterable, start=0)`

Return an enumerate object. `iterable` must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from `start` which defaults to 0) and the values obtained from iterating over `iterable`.

```
1 >>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
2 >>> list(enumerate(seasons))
3 [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
4 >>> list(enumerate(seasons, start=1))
5 [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
1 def enumerate(sequence, start=0):
2 n = start
3 for elem in sequence:
4 yield n, elem
5 n += 1
```

```
eval(expression[, globals[, locals]])
```

The arguments are a string and optional `globals` and `locals`. If provided, `globals` must be a dictionary. If provided, `locals` can be any mapping object.

The `expression` argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the `globals` and `locals` dictionaries as global and local namespace. If the `globals` dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before `expression` is parsed. This means that `expression` normally has full access to

the standard `builtins` module and restricted environments are propagated. If the `locals` dictionary is omitted it defaults to the `globals` dictionary. If both dictionaries are omitted, the expression is executed with the `globals` and `locals` in the environment where `eval()` is called. Note, `eval()` does not have access to the nested scopes (non-locals) in the enclosing environment.

The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:>>>

```
1 >>> x = 1
2 >>> eval('x+1')
3 2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with `'exec'` as the `mode` argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

`exec (object[, globals[, locals]])`

This function supports dynamic execution of Python code. `object` must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).<sup>1</sup> If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section “File input” in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If exec gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `exec()`.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

#### Note

The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

#### Note

The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns. `filter(function, iterable)`

Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression  
`(item for item in iterable if function(item))` if *function* is not `None` and  
`(item for item in iterable if item)` if *function* is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of `iterable` for which `function` returns false.

`class float ([x])`

Return a floating point number constructed from a number or string `x`.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
1 sign ::= "+" | "-"
2 infinity ::= "Infinity" | "inf"
3 nan ::= "nan"
4 numeric_value ::= floatnumber | infinity | nan
5 numeric_string ::= [sign] numeric_value
```

Here `floatnumber` is the form of a Python floating-point literal, described in Floating point literals. Case is not significant, so, for example, “`inf`”, “`Inf`”, “`INFINITY`” and “`iNfINity`” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If no argument is given, `0.0` is returned.

Examples:>>>

```
1 >>> float('+1.23')
2 1.23
3 >>> float(' -12345\n')
4 -12345.0
5 >>> float('1e-003')
6 0.001
```

```
7 >>> float('+1E6')
8 1000000.0
9 >>> float('-Infinity')
10 -inf
```

The float type is described in Numeric Types – int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__float__()` is not defined. `format(value[, format_spec])`

Convert a *value* to a “formatted” representation, as controlled by *format\_spec*. The interpretation of *format\_spec* will depend on the type of the *value* argument, however there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

The default *format\_spec* is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to

`type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value’s `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the *format\_spec* is non-empty, or if either the *format\_spec* or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises `TypeError` if *format\_spec* is not an empty string.`class frozenset ([iterable])`

Return a new `frozenset` object, optionally with elements taken from *iterable*. `frozenset` is a built-in class. See `frozenset` and Set Types – set, frozenset for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module. `getattr(object, name[, default])`

Return the value of the named attribute of `object`. `name` must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

#### Note

Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to retrieve it with `getattr() . globals()`

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called). `hasattr(object, name)`

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.) `hash(object)`

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

#### Note

For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine. See `__hash__()` for details. `help([object])`

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function, when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ

entry on positional-only parameters.

This function is added to the built-in namespace by the `site` module.

Changed in version 3.4: Changes to `pydoc` and `inspect` mean that the reported signatures for callables are now more comprehensive and consistent. `hex(x)`

Convert an integer number to a lowercase hexadecimal string prefixed with “0x”. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:>>>

```
1 >>> hex(255)
2 '0xff'
3 >>> hex(-42)
4 '-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:>>>

```
1 >>> '%#x' % 255, '%x' % 255, '%X' % 255
2 ('0xff', 'ff', 'FF')
3 >>> format(255, '#x'), format(255, 'x'), format(255, 'X')
4 ('0xff', 'ff', 'FF')
5 >>> f'{255:#x}', f'{255:x}', f'{255:X}'
6 ('0xff', 'ff', 'FF')
```

See also `format()` for more information.

See also `int()` for converting a hexadecimal string to an integer using a base of 16.

Note

To obtain a hexadecimal string representation for a float, use the `float.hex()` method. `id(object)`

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may

have the same `id()` value.

**CPython implementation detail:** This is the address of the object in memory.

Raises an auditing event `builtins.id` with argument `id . input ([prompt])`

If the `prompt` argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:>>>

```
1 >>> s = input('--> ')
2 --> Monty Python's Flying Circus
3 >>> s
4 "Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Raises an auditing event `builtins.input` with argument `prompt` before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

`class int ([x])class int (x, base=10)`

Return an integer object constructed from a number or string `x`, or return `0` if no arguments are given. If `x` defines `__int__()`, `int(x)` returns `x.__int__()`. If `x` defines `__index__()`, it returns `x.__index__()`. If `x` defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If `x` is not a number or if `base` is given, then `x` must be a string, `bytes`, or `bytearray` instance representing an integer literal in radix `base`. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `z`) having values 10 to 35. The default `base` is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b` / `0B`, `0o` / `0O`, or `0x` / `0X`, as with integer literals in code. Base 0 means to interpret exactly

as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in Numeric Types – int, float, complex.

Changed in version 3.4: If `base` is not an instance of `int` and the `base` object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: `x` is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined. `isinstance(object, classinfo)`

Return `True` if the `object` argument is an instance of the `classinfo` argument, or of a (direct, indirect or virtual) subclass thereof. If `object` is not an object of the given type, the function always returns `False`. If `classinfo` is a tuple of type objects (or recursively, other such tuples), return `True` if `object` is an instance of any of the types. If `classinfo` is not a type or tuple of types and such tuples, a `TypeError` exception is raised. `issubclass(class, classinfo)`

Return `True` if `class` is a subclass (direct, indirect or virtual) of `classinfo`. A class is considered a subclass of itself. `classinfo` may be a tuple of class objects, in which case every entry in `classinfo` will be checked. In any other case, a `TypeError` exception is raised. `iter(object[, sentinel])`

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, `object` must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, `sentinel`, is given, then `object` must be a callable object. The iterator created in this case will call `object` with no arguments for each call to its `__next__()` method; if the value returned is equal to `sentinel`, `StopIteration` will be raised, otherwise the value will be returned.

See also [Iterator Types](#).

One useful application of the second form of `iter()` is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
1 from functools import partial
2 with open('mydata.db', 'rb') as f:
3 for block in iter(partial(f.read, 64), b''):
4 process_block(block)
```

`len (s)`

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

**C**Python implementation detail: `len` raises `OverflowError` on lengths larger than `sys.maxsize`, such as `range(2 ** 100)`.

Rather than being a function, `list` is actually a mutable sequence type, as documented in Lists and Sequence Types – `list, tuple, range, locals ()`

Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks. Note that at the module level, `locals()` and `globals()` are the same dictionary.

Note

The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter. `map (function, iterable, ...)`

Return an iterator that applies `function` to every item of `iterable`, yielding the results. If additional `iterable` arguments are passed, `function` must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

`max (iterable, *, key, default)`

`max (arg1, arg2, *args[, key])`

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The `key` argument specifies a one-argument ordering function like that used for `list.sort()`. The `default` argument specifies an object to return if the provided iterable is empty. If the iterable is empty and `default` is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as

```
sorted(iterable, key=keyfunc, reverse=True)[0] and
heapq.nlargest(1, iterable, key=keyfunc).
```

New in version 3.4: The `default` keyword-only argument.

Changed in version 3.8: The `key` can be `None` .`class` `memoryview` (`object`)

Return a “memory view” object created from the given argument. See Memory Views for more information. `min(iterable, *, key, default)` `min(arg1, arg2, *args[, key])`

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The `key` argument specifies a one-argument ordering function like that used for `list.sort()`. The `default` argument specifies an object to return if the provided iterable is empty. If the iterable is empty and `default` is not provided, a `ValueError` is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

New in version 3.4: The `default` keyword-only argument.

Changed in version 3.8: The `key` can be `None` . `next(iterator[, default])`

Retrieve the next item from the *iterator* by calling its `__next__()` method. If `default` is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

Return a new featureless object. `object` is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

## Note

`object` does *not* have a `__dict__`, so you can't assign arbitrary attributes to an instance of the `object` class. `oct(x)`

Convert an integer number to an octal string prefixed with “0o”. The result is a valid Python expression. If `x` is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:>>>

```
1 >>> oct(8)
2 '0o10'
3 >>> oct(-56)
4 '-0o70'
```

If you want to convert an integer number to octal string either with prefix “0o” or not, you can use either of the following ways.>>>

```
1 >>> '%#o' % 10, '%o' % 10
2 ('0o12', '12')
3 >>> format(10, '#o'), format(10, 'o')
4 ('0o12', '12')
5 >>> f'{10:#o}', f'{10:o}'
6 ('0o12', '12')
```

See also `format()` for more information.

>

`open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`

Open `file` and return a corresponding file object. If the file cannot be opened, an  `OSError`  is raised. See [Reading and Writing Files](#) for more examples of how to use this function.

`file` is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless `closefd` is set to `False`.)

`mode` is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if `encoding` is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave `encoding` unspecified.) The available modes are:

Character	Meaning
<code>'r'</code>	open for reading (default)
<code>'w'</code>	open for writing, truncating the file first
<code>'x'</code>	open for exclusive creation, failing if the file already exists
<code>'a'</code>	open for writing, appending to the end of the file if it exists
<code>'b'</code>	binary mode
<code>'t'</code>	text mode (default)
<code>'+'</code>	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the `mode` argument) return contents as `bytes` objects without any decoding. In text mode (the default, or when `'t'` is included in the `mode`

argument), the contents of the file are returned as `str`, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

There is an additional mode character permitted, `'u'`, which no longer has any effect, and is considered deprecated. It previously enabled universal newlines in text mode, which became the default behaviour in Python 3.0. Refer to the documentation of the `newline` parameter for further details.

#### Note

Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

*buffering* is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files.

*encoding* is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any text encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

*errors* is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers), though any error handling name that has been registered with `codecs.register_error()` is also valid. The standard names include:

- `'strict'` to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.

- 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data.
- 'surrogateescape' will represent any incorrect bytes as low surrogate code units ranging from U+DC80 to U+DCFF. These surrogate code units will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- 'xmlcharrefreplace' is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
- 'backslashreplace' replaces malformed data by Python's backslashed escape sequences.
- 'namereplace' (also only supported when writing) replaces unsupported characters with `\N{...}` escape sequences.

`newline` controls how universal newlines mode works (it only applies to text mode). It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if `newline` is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If `newline` is `''` or `'\n'`, no translation takes place. If `newline` is any of the other legal values, any `'\n'` characters written are translated to the given string.

If `closefd` is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` must be `True` (the default) otherwise an error will be raised.

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).

The newly created file is non-inheritable.

The following example uses the `dir_fd` parameter of the `os.open()` function to open a file relative to a given directory:>>>

```
1 >>> import os
2 >>> dir_fd = os.open('somedir', os.O_RDONLY)
3 >>> def opener(path, flags):
4 ... return os.open(path, flags, dir_fd=dir_fd)
5 ...
6 >>> with open('spamspam.txt', 'w', opener=opener) as f:
7 ... print('This will be written to somedir/spamspam.txt', file=f)
8 ...
9 >>> os.close(dir_fd) # don't leak a file descriptor
```

The type of file object returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode ( `'w'` , `'r'` , `'wt'` , `'rt'` , etc.), it returns a subclass of `io.TextIOBase` (specifically `io.TextIOWrapper` ). When used to open a file in a binary mode with buffering, the returned class is a subclass of `io.BufferedIOBase` . The exact class varies: in read binary mode, it returns an `io.BufferedReader` ; in write binary and append binary modes, it returns an `io.BufferedWriter` , and in read/write mode, it returns an `io.BufferedRandom` . When buffering is disabled, the raw stream, a subclass of `io.RawIOBase` , `io.FileIO` , is returned.

See also the file handling modules, such as, `fileinput` , `io` (where `open()` is declared), `os` , `os.path` , `tempfile` , and `shutil` .

Raises an auditing event `open` with arguments `file` , `mode` , `flags` .

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The `opener` parameter was added.
- The `'x'` mode was added.
- `IOError` used to be raised, it is now an alias of `OSError` .
- `FileExistsError` is now raised if the file opened in exclusive creation mode ( `'x'` ) already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Deprecated since version 3.4, will be removed in version 3.10: The `'u'` mode.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).
- The `'namereplace'` error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing `os.PathLike`.
- On Windows, opening a console buffer may return a subclass of `io.RawIOBase` other than `io.FileIO`.

`ord(c)`

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of `chr()`. `pow(base, exp[, mod])`

Return `base` to the power `exp`; if `mod` is present, return `base` to the power `exp`, modulo `mod` (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`.

For `int` operands `base` and `exp`, if `mod` is present, `mod` must also be of integer type and `mod` must be nonzero. If `mod` is present and `exp` is negative, `base` must be relatively prime to `mod`. In that case, `pow(inv_base, -exp, mod)` is returned, where `inv_base` is an inverse to `base` modulo `mod`.

Here's an example of computing an inverse for 38 modulo 97 :>>>

```
1 >>> pow(38, -1, mod=97)
2 23
3 >>> 23 * 38 % 97 == 1
4 True
```

Changed in version 3.8: For `int` operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.8: Allow keyword arguments. Formerly, only positional arguments were supported. `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument. `class property (fget=None, fset=None, fdel=None, doc=None)`

Return a property attribute.

*fget* is a function for getting an attribute value. *fset* is a function for setting an attribute value. *fdel* is a function for deleting an attribute value. And *doc* creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
1 class C:
2 def __init__(self):
3 self._x = None
4
5 def getx(self):
6 return self._x
7
8 def setx(self, value):
9 self._x = value
10
11 def delx(self):
12 del self._x
13
14 x = property(getx, setx, delx, "I'm the 'x' property.")
```

If `c` is an instance of `C`, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

If given, `doc` will be the docstring of the property attribute. Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a decorator:

```
1 class Parrot:
2 def __init__(self):
3 self._voltage = 100000
4
5 @property
6 def voltage(self):
7 """Get the current voltage."""
8 return self._voltage
```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for `voltage` to “Get the current voltage.”

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```

1 class C:
2 def __init__(self):
3 self._x = None
4
5 @property
6 def x(self):
7 """I'm the 'x' property."""
8 return self._x
9
10 @x.setter
11 def x(self, value):
12 self._x = value
13
14 @x.deleter
15 def x(self):
16 del self._x

```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

`class range (stop)` `range (start, stop[, step])`

Rather than being a function, `range` is actually an immutable sequence type, as documented in Ranges and Sequence Types – list, tuple, range. `repr (object)`

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. `reversed (seq)`

Return a reverse iterator. `seq` must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`). `round (number[, ndigits])`

Return `number` rounded to `ndigits` precision after the decimal point. If `ndigits` is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus `ndigits`; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for `ndigits` (positive, zero, or negative). The return value is an integer if `ndigits` is omitted or `None`. Otherwise the return value has the same type as `number`.

For a general Python object `number`, `round` delegates to `number.__round__`.

#### Note

The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating Point Arithmetic: Issues and Limitations for more information.

Return a new `set` object, optionally with elements taken from `iterable`. `set` is a built-in class. See `set` and Set Types – `set`, `frozenset` for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

#### Note

Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with `setattr()`.

Return a slice object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They

have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator. `sorted(iterable, *, key=None, reverse=False)`

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

`key` specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style `cmp` function to a `key` function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal – this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [Sorting HOW TO](#). `@staticmethod`

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
1 class C:
2 @staticmethod
3 def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function decorator – see [Function definitions](#) for details.

A static method can be called either on the class (such as `c.f()`) or on an instance (such as `c().f()`).

Static methods in Python are similar to those found in Java or C++. Also see `classmethod()` for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
1 class C:
2 builtin_open = staticmethod(open)
```

For more information on static methods, see The standard type hierarchy.`class str` (`object="`)`class str (object=b", encoding='utf-8', errors='strict')`

Return a `str` version of `object`. See `str()` for details.

`str` is the built-in string class. For general information about strings, see Text Sequence Type  
— `str.sum (iterable, /, start=0)`

Sums `start` and the items of an `iterable` from left to right and returns the total. The `iterable`'s items are normally numbers, and the `start` value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `'.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

Changed in version 3.8: The `start` parameter can be specified as a keyword argument. `super([type], object-or-type)]`

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class.

The `object-or-type` determines the method resolution order to be searched. The search starts from the class right after the `type`.

For example, if `__mro__` of *object-or-type* is `D -> B -> C -> A -> object` and the value of `type` is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the *object-or-type* lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
1 class C(B):
2 def method(self, arg):
3 super().method(arg) # This does the same thing as:
4 # super(C, self).method(arg)
```

In addition to method lookups, `super()` also works for attribute lookups. One possible use case for this is calling descriptors in a parent or sibling class.

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports

cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super().__name__`.

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see guide to using `super().__class__ tuple ([iterable])`

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in [Tuples and Sequence Types – list, tuple, range](#).`__class__` type (`object`)`__class__` type (`name, bases, dict, **kwds`)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute; if empty, `object`, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical `type` objects:>>>

```
1 >>> class X:
2 ... a = 1
3 ...
4 >>> X = type('X', (), dict(a=1))
```

See also [Type Objects](#).

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

See also [Customizing class creation](#).

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object. `vars([object])`

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute). `zip(*iterables)`

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```
1 def zip(*iterables):
2 # zip('ABCD', 'xy') --> Ax By
3 sentinel = object()
4 iterators = [iter(it) for it in iterables]
5 while iterators:
6 result = []
7 for it in iterators:
8 elem = next(it, sentinel)
9 if elem is sentinel:
10 return
11 result.append(elem)
```

```
12 yield tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

`zip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:>>>

```
1 >>> x = [1, 2, 3]
2 >>> y = [4, 5, 6]
3 >>> zipped = zip(x, y)
4 >>> list(zipped)
5 [(1, 4), (2, 5), (3, 6)]
6 >>> x2, y2 = zip(*zip(x, y))
7 >>> x == list(x2) and y == list(y2)
8 True
```

```
__import__(name, globals=None, locals=None, fromlist=(), level=0)
```

## Note

This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see [PEP 302](#)) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the `import` statement.

*level* specifies whether to use absolute or relative imports. `0` (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see [PEP 328](#) for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [])
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
1 _temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
2 eggs = _temp.eggs
3 saus = _temp.sausage
```

practice

# Exercises

1. Write a Python program to print the following string in a specific format (see the output). Go to the editor

*Sample String : "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are"* *Output :*

```
1 Twinkle, twinkle, little star,
2 How I wonder what you are!
3 Up above the world so high,
4 Like a diamond in the sky.
5 Twinkle, twinkle, little star,
6 How I wonder what you are
```

[Click me to see the sample solution](#)

2. Write a Python program to get the Python version you are using. Go to the editor

[Click me to see the sample solution](#)

3. Write a Python program to display the current date and time.

*Sample Output :*

Current date and time :

2014-07-05 14:34:14

[Click me to see the sample solution](#)

4. Write a Python program which accepts the radius of a circle from the user and compute the area. Go to the editor

*Sample Output :*

r = 1.1

Area = 3.8013271108436504

[Click me to see the sample solution](#)

5. Write a Python program which accepts the user's first and last name and print them in reverse order with a space between them. Go to the editor

[Click me to see the sample solution](#)

**6.** Write a Python program which accepts a sequence of comma-separated numbers from user and generate a list and a tuple with those numbers. Go to the editor

*Sample data : 3, 5, 7, 23*

*Output :*

List : [3, 5, 7, 23]

Tuple : (3, 5, 7, 23)

[Click me to see the sample solution](#)

**7.** Write a Python program to accept a filename from the user and print the extension of that.

[Go to the editor](#)

*Sample filename : abc.java*

*Output : java*

[Click me to see the sample solution](#)

**8.** Write a Python program to display the first and last colors from the following list. Go to the editor

color\_list = ["Red","Green","White", "Black"]

[Click me to see the sample solution](#)

**9.** Write a Python program to display the examination schedule. (extract the date from exam\_st\_date). Go to the editor

exam\_st\_date = (11, 12, 2014)

Sample Output : The examination will start from : 11 / 12 / 2014

[Click me to see the sample solution](#)

**10.** Write a Python program that accepts an integer (n) and computes the value of n+nn+nnn.

[Go to the editor](#)

*Sample value of n is 5*

*Expected Result : 615*

[Click me to see the sample solution](#)

**11.** Write a Python program to print the documents (syntax, description etc.) of Python built-in function(s).

*Sample function : abs()*

*Expected Result :*

abs(number) -> number

Return the absolute value of the argument.

[Click me to see the sample solution](#)

**12.** Write a Python program to print the calendar of a given month and year.

*Note :* Use 'calendar' module.

[Click me to see the sample solution](#)

**13.** Write a Python program to print the following 'here document'. Go to the editor

*Sample string :*

a string that you "don't" have to escape

This

is a ..... multi-line

heredoc string ----> example

[Click me to see the sample solution](#)

**14.** Write a Python program to calculate number of days between two dates.

*Sample dates :* (2014, 7, 2), (2014, 7, 11)

*Expected output :* 9 days

[Click me to see the sample solution](#)

**15.** Write a Python program to get the volume of a sphere with radius 6.

[Click me to see the sample solution](#)

**16.** Write a Python program to get the difference between a given number and 17, if the number is greater than 17 return double the absolute difference. Go to the editor

[Click me to see the sample solution](#)

**17.** Write a Python program to test whether a number is within 100 of 1000 or 2000. Go to the editor

[Click me to see the sample solution](#)

**18.** Write a Python program to calculate the sum of three given numbers, if the values are equal then return three times of their sum. Go to the editor

[Click me to see the sample solution](#)

**19.** Write a Python program to get a new string from a given string where "Is" has been added to the front. If the given string already begins with "Is" then return the string unchanged. Go to the editor

[Click me to see the sample solution](#)

**20.** Write a Python program to get a string which is n (non-negative integer) copies of a given string. Go to the editor

[Click me to see the sample solution](#)

**21.** Write a Python program to find whether a given number (accept from the user) is even or odd, print out an appropriate message to the user. Go to the editor

[Click me to see the sample solution](#)

**22.** Write a Python program to count the number 4 in a given list. Go to the editor

[Click me to see the sample solution](#)

**23.** Write a Python program to get the n (non-negative integer) copies of the first 2 characters of a given string. Return the n copies of the whole string if the length is less than 2. Go to the editor

[Click me to see the sample solution](#)

**24.** Write a Python program to test whether a passed letter is a vowel or not. Go to the editor

[Click me to see the sample solution](#)

**25.** Write a Python program to check whether a specified value is contained in a group of values. Go to the editor

*Test Data :*

3 -> [1, 5, 8, 3] : True

-1 -> [1, 5, 8, 3] : False

[Click me to see the sample solution](#)

**26.** Write a Python program to create a histogram from a given list of integers. Go to the editor

[Click me to see the sample solution](#)

**27.** Write a Python program to concatenate all elements in a list into a string and return it. Go to the editor

[Click me to see the sample solution](#)

**28.** Write a Python program to print all even numbers from a given numbers list in the same order and stop the printing if any numbers that come after 237 in the sequence. Go to the editor

*Sample numbers list :*

```
1 numbers = [
2 386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953,
3 399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687
```

```
4 815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 741
5 958, 743, 527
6]
```

[Click me to see the sample solution](#)

**29.** Write a Python program to print out a set containing all the colors from color\_list\_1 which are not present in color\_list\_2. Go to the editor

*Test Data :*

```
color_list_1 = set(["White", "Black", "Red"])
color_list_2 = set(["Red", "Green"])
```

*Expected Output :*

```
{'Black', 'White'}
```

[Click me to see the sample solution](#)

**30.** Write a Python program that will accept the base and height of a triangle and compute the area. Go to the editor

[Click me to see the sample solution](#)

**31.** Write a Python program to compute the greatest common divisor (GCD) of two positive integers. Go to the editor

[Click me to see the sample solution](#)

**32.** Write a Python program to get the least common multiple (LCM) of two positive integers.

[Go to the editor](#)

[Click me to see the sample solution](#)

**33.** Write a Python program to sum of three given integers. However, if two values are equal sum will be zero. Go to the editor

[Click me to see the sample solution](#)

**34.** Write a Python program to sum of two given integers. However, if the sum is between 15 to 20 it will return 20. Go to the editor

[Click me to see the sample solution](#)

**35.** Write a Python program that will return true if the two given integer values are equal or their sum or difference is 5. Go to the editor

[Click me to see the sample solution](#)

**36.** Write a Python program to add two objects if both objects are an integer type. Go to the editor

[Click me to see the sample solution](#)

**37.** Write a Python program to display your details like name, age, address in three different lines. Go to the editor

[Click me to see the sample solution](#)

**38.** Write a Python program to solve  $(x + y) * (x + y)$ . Go to the editor

*Test Data :*  $x = 4, y = 3$

*Expected Output :*  $(4 + 3)^2 = 49$

[Click me to see the sample solution](#)

**39.** Write a Python program to compute the future value of a specified principal amount, rate of interest, and a number of years. Go to the editor

*Test Data :*  $amt = 10000, int = 3.5, years = 7$

*Expected Output :* 12722.79

[Click me to see the sample solution](#)

**40.** Write a Python program to compute the distance between the points  $(x_1, y_1)$  and  $(x_2, y_2)$ .

[Go to the editor](#)

[Click me to see the sample solution](#)

**41.** Write a Python program to check whether a file exists. Go to the editor

[Click me to see the sample solution](#)

**42.** Write a Python program to determine if a Python shell is executing in 32bit or 64bit mode on OS. Go to the editor

[Click me to see the sample solution](#)

**43.** Write a Python program to get OS name, platform and release information. Go to the editor

[Click me to see the sample solution](#)

**44.** Write a Python program to locate Python site-packages. Go to the editor

[Click me to see the sample solution](#)

**45.** Write a python program to call an external command in Python. Go to the editor

[Click me to see the sample solution](#)

**46.** Write a python program to get the path and name of the file that is currently executing. Go to the editor

[Click me to see the sample solution](#)

**47.** Write a Python program to find out the number of CPUs using. Go to the editor

[Click me to see the sample solution](#)

**48.** Write a Python program to parse a string to Float or Integer. Go to the editor

[Click me to see the sample solution](#)

**49.** Write a Python program to list all files in a directory in Python. Go to the editor

[Click me to see the sample solution](#)

**50.** Write a Python program to print without newline or space. Go to the editor

[Click me to see the sample solution](#)

**51.** Write a Python program to determine profiling of Python programs. Go to the editor

Note: A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.

[Click me to see the sample solution](#)

**52.** Write a Python program to print to stderr. Go to the editor

[Click me to see the sample solution](#)

**53.** Write a python program to access environment variables. Go to the editor

[Click me to see the sample solution](#)

**54.** Write a Python program to get the current username Go to the editor

[Click me to see the sample solution](#)

**55.** Write a Python to find local IP addresses using Python's stdlib Go to the editor

[Click me to see the sample solution](#)

**56.** Write a Python program to get height and width of the console window. Go to the editor

[Click me to see the sample solution](#)

**57.** Write a Python program to get execution time for a Python method. Go to the editor

[Click me to see the sample solution](#)

**58.** Write a Python program to sum of the first n positive integers. Go to the editor

[Click me to see the sample solution](#)

**59.** Write a Python program to convert height (in feet and inches) to centimeters. Go to the editor

[Click me to see the sample solution](#)

**60.** Write a Python program to calculate the hypotenuse of a right angled triangle. Go to the editor

[Click me to see the sample solution](#)

**61.** Write a Python program to convert the distance (in feet) to inches, yards, and miles. Go to the editor

[Click me to see the sample solution](#)

**62.** Write a Python program to convert all units of time into seconds. Go to the editor

[Click me to see the sample solution](#)

**63.** Write a Python program to get an absolute file path. Go to the editor

[Click me to see the sample solution](#)

**64.** Write a Python program to get file creation and modification date/times. Go to the editor

[Click me to see the sample solution](#)

**65.** Write a Python program to convert seconds to day, hour, minutes and seconds. Go to the editor

[Click me to see the sample solution](#)

**66.** Write a Python program to calculate body mass index. Go to the editor

[Click me to see the sample solution](#)

**67.** Write a Python program to convert pressure in kilopascals to pounds per square inch, a millimeter of mercury (mmHg) and atmosphere pressure. Go to the editor

[Click me to see the sample solution](#)

**68.** Write a Python program to calculate the sum of the digits in an integer. Go to the editor

[Click me to see the sample solution](#)

**69.** Write a Python program to sort three integers without using conditional statements and loops. Go to the editor

[Click me to see the sample solution](#)

**70.** Write a Python program to sort files by date. Go to the editor

[Click me to see the sample solution](#)

**71.** Write a Python program to get a directory listing, sorted by creation date. Go to the editor

[Click me to see the sample solution](#)

**72.** Write a Python program to get the details of math module. Go to the editor

[Click me to see the sample solution](#)

**73.** Write a Python program to calculate midpoints of a line. Go to the editor

[Click me to see the sample solution](#)

**74.** Write a Python program to hash a word. Go to the editor

[Click me to see the sample solution](#)

**75.** Write a Python program to get the copyright information and write Copyright information in Python code. Go to the editor

[Click me to see the sample solution](#)

**76.** Write a Python program to get the command-line arguments (name of the script, the number of arguments, arguments) passed to a script. Go to the editor

[Click me to see the sample solution](#)

**77.** Write a Python program to test whether the system is a big-endian platform or little-endian platform. Go to the editor

[Click me to see the sample solution](#)

**78.** Write a Python program to find the available built-in modules. Go to the editor

[Click me to see the sample solution](#)

**79.** Write a Python program to get the size of an object in bytes. Go to the editor

[Click me to see the sample solution](#)

**80.** Write a Python program to get the current value of the recursion limit. Go to the editor

[Click me to see the sample solution](#)

**81.** Write a Python program to concatenate N strings. Go to the editor

[Click me to see the sample solution](#)

**82.** Write a Python program to calculate the sum of all items of a container (tuple, list, set, dictionary). Go to the editor

[Click me to see the sample solution](#)

**83.** Write a Python program to test whether all numbers of a list is greater than a certain number. Go to the editor

[Click me to see the sample solution](#)

**84.** Write a Python program to count the number occurrence of a specific character in a string. Go to the editor

[Click me to see the sample solution](#)

**85.** Write a Python program to check whether a file path is a file or a directory. Go to the editor

[Click me to see the sample solution](#)

**86.** Write a Python program to get the ASCII value of a character. Go to the editor

[Click me to see the sample solution](#)

**87.** Write a Python program to get the size of a file. Go to the editor

[Click me to see the sample solution](#)

**88.** Given variables  $x=30$  and  $y=20$ , write a Python program to print "30+20=50". Go to the editor

Given a variable name, if the value is 1, display the string "First day of a Month!" and do nothing if the value is not equal.

[Click me to see the sample solution](#)

**89.** Write a Python program to perform an action if a condition is true. Go to the editor

Given a variable name, if the value is 1, display the string "First day of a Month!" and do nothing if the value is not equal.

[Click me to see the sample solution](#)

**90.** Write a Python program to create a copy of its own source code. Go to the editor

[Click me to see the sample solution](#)

**91.** Write a Python program to swap two variables. Go to the editor

[Click me to see the sample solution](#)

**92.** Write a Python program to define a string containing special characters in various forms.

[Go to the editor](#)

[Click me to see the sample solution](#)

**93.** Write a Python program to get the Identity, Type, and Value of an object. Go to the editor

[Click me to see the sample solution](#)

**94.** Write a Python program to convert a byte string to a list of integers. Go to the editor

[Click me to see the sample solution](#)

**95.** Write a Python program to check whether a string is numeric. Go to the editor

[Click me to see the sample solution](#)

**96.** Write a Python program to print the current call stack. Go to the editor

[Click me to see the sample solution](#)

**97.** Write a Python program to list the special variables used within the language. Go to the editor

[Click me to see the sample solution](#)

**98.** Write a Python program to get the system time. Go to the editor

Note : The system time is important for debugging, network information, random number seeds, or something as simple as program performance.

[Click me to see the sample solution](#)

**99.** Write a Python program to clear the screen or terminal. Go to the editor

[Click me to see the sample solution](#)

**100.** Write a Python program to get the name of the host on which the routine is running. Go to the editor

[Click me to see the sample solution](#)

**101.** Write a Python program to access and print a URL's content to the console. Go to the editor

[Click me to see the sample solution](#)

**102.** Write a Python program to get system command output. Go to the editor

[Click me to see the sample solution](#)

**103.** Write a Python program to extract the filename from a given path. Go to the editor

[Click me to see the sample solution](#)

**104.** Write a Python program to get the effective group id, effective user id, real group id, a list of supplemental group ids associated with the current process. Go to the editor

Note: Availability: Unix.

[Click me to see the sample solution](#)

**105.** Write a Python program to get the users environment. Go to the editor

[Click me to see the sample solution](#)

**106.** Write a Python program to divide a path on the extension separator. Go to the editor

[Click me to see the sample solution](#)

**107.** Write a Python program to retrieve file properties. Go to the editor

[Click me to see the sample solution](#)

**108.** Write a Python program to find path refers to a file or directory when you encounter a path name. Go to the editor

[Click me to see the sample solution](#)

**109.** Write a Python program to check if a number is positive, negative or zero. Go to the editor

[Click me to see the sample solution](#)

**110.** Write a Python program to get numbers divisible by fifteen from a list using an anonymous function. Go to the editor

[Click me to see the sample solution](#)

**111.** Write a Python program to make file lists from current directory using a wildcard. Go to the editor

[Click me to see the sample solution](#)

**112.** Write a Python program to remove the first item from a specified list. Go to the editor

[Click me to see the sample solution](#)

**113.** Write a Python program to input a number, if it is not a number generates an error message. Go to the editor

[Click me to see the sample solution](#)

**114.** Write a Python program to filter the positive numbers from a list. Go to the editor

[Click me to see the sample solution](#)

**115.** Write a Python program to compute the product of a list of integers (without using for loop). Go to the editor

[Click me to see the sample solution](#)

**116.** Write a Python program to print Unicode characters. Go to the editor

[Click me to see the sample solution](#)

**117.** Write a Python program to prove that two string variables of same value point same memory location. Go to the editor

[Click me to see the sample solution](#)

**118.** Write a Python program to create a bytearray from a list. Go to the editor

[Click me to see the sample solution](#)

**119.** Write a Python program to round a floating-point number to specified number decimal places. Go to the editor

[Click me to see the sample solution](#)

**120.** Write a Python program to format a specified string limiting the length of a string. Go to the editor

[Click me to see the sample solution](#)

**121.** Write a Python program to determine whether variable is defined or not. Go to the editor

[Click me to see the sample solution](#)

**122.** Write a Python program to empty a variable without destroying it. Go to the editor

Sample data: n=20

d = {"x":200}

Expected Output : 0

{}

[Click me to see the sample solution](#)

**123.** Write a Python program to determine the largest and smallest integers, longs, floats. Go to the editor

[Click me to see the sample solution](#)

**124.** Write a Python program to check whether multiple variables have the same value. Go to the editor

[Click me to see the sample solution](#)

**125.** Write a Python program to sum of all counts in a collections. Go to the editor

[Click me to see the sample solution](#)

**126.** Write a Python program to get the actual module object for a given object. Go to the editor

[Click me to see the sample solution](#)

**127.** Write a Python program to check whether an integer fits in 64 bits. Go to the editor

[Click me to see the sample solution](#)

**128.** Write a Python program to check whether lowercase letters exist in a string. Go to the editor

editor

[Click me to see the sample solution](#)

**129.** Write a Python program to add leading zeroes to a string. Go to the editor

[Click me to see the sample solution](#)

**130.** Write a Python program to use double quotes to display strings. Go to the editor

[Click me to see the sample solution](#)

**131.** Write a Python program to split a variable length string into variables. Go to the editor

[Click me to see the sample solution](#)

**132.** Write a Python program to list home directory without absolute path. Go to the editor

[Click me to see the sample solution](#)

**133.** Write a Python program to calculate the time runs (difference between start and current time) of a program. Go to the editor

[Click me to see the sample solution](#)

**134.** Write a Python program to input two integers in a single line. Go to the editor

[Click me to see the sample solution](#)

**135.** Write a Python program to print a variable without spaces between values. Go to the editor

Sample value : x =30

Expected output : Value of x is "30"

[Click me to see the sample solution](#)

**136.** Write a Python program to find files and skip directories of a given directory. Go to the editor

[Click me to see the sample solution](#)

**137.** Write a Python program to extract single key-value pair of a dictionary in variables. Go to the editor

[Click me to see the sample solution](#)

**138.** Write a Python program to convert true to 1 and false to 0. Go to the editor

[Click me to see the sample solution](#)

**139.** Write a Python program to valid a IP address. Go to the editor

[Click me to see the sample solution](#)

**140.** Write a Python program to convert an integer to binary keep leading zeros. Go to the editor

Sample data : x=12

Expected output : 00001100

0000001100

[Click me to see the sample solution](#)

**141.** Write a python program to convert decimal to hexadecimal. Go to the editor

Sample decimal number: 30, 4

Expected output: 1e, 04

[Click me to see the sample solution](#)

**142.** Write a Python program to find the operating system name, platform and platform release date. Go to the editor

Operating system name:

posix

Platform name:

Linux

Platform release:

4.4.0-47-generic

[Click me to see the sample solution](#)

**143.** Write a Python program to determine if the python shell is executing in 32bit or 64bit mode on operating system. Go to the editor

[Click me to see the sample solution](#)

**144.** Write a Python program to check whether variable is integer or string. Go to the editor

[Click me to see the sample solution](#)

**145.** Write a Python program to test if a variable is a list or tuple or a set. Go to the editor

[Click me to see the sample solution](#)

**146.** Write a Python program to find the location of Python module sources. Go to the editor

[Click me to see the sample solution](#)

**147.** Write a Python function to check whether a number is divisible by another number. Accept two integers values form the user. Go to the editor

[Click me to see the sample solution](#)

**148.** Write a Python function to find the maximum and minimum numbers from a sequence of numbers. Go to the editor

Note: Do not use built-in functions.

[Click me to see the sample solution](#)

**149.** Write a Python function that takes a positive integer and returns the sum of the cube of all the positive integers smaller than the specified number. Go to the editor

[Click me to see the sample solution](#)

**150.** Write a Python function to check whether a distinct pair of numbers whose product is odd present in a sequence of integer values. Go to the editor

[Click me to see the sample solution](#)

# Exercises



PythonPracticeGists

<https://replit.com/@bgoonz/Python-Practice-Gists>

\*\*\*\*

**Long list of small examples at bottom of page...**

\*\*\*\*

1. Write a Python program to print the following string in a specific format (see the output). Go to the editor

*Sample String : "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are" Output :*

```
1 Twinkle, twinkle, little star,
2 How I wonder what you are!
3 Up above the world so high,
4 Like a diamond in the sky.
5 Twinkle, twinkle, little star,
6 How I wonder what you are
```

Click me to see the sample solution

2. Write a Python program to get the Python version you are using. Go to the editor

Click me to see the sample solution

3. Write a Python program to display the current date and time.

*Sample Output :*

Current date and time :

2014-07-05 14:34:14

Click me to see the sample solution

4. Write a Python program which accepts the radius of a circle from the user and compute the area. Go to the editor

*Sample Output :*

r = 1.1

Area = 3.8013271108436504

[Click me to see the sample solution](#)

**5.** Write a Python program which accepts the user's first and last name and print them in reverse order with a space between them. Go to the editor

[Click me to see the sample solution](#)

**6.** Write a Python program which accepts a sequence of comma-separated numbers from user and generate a list and a tuple with those numbers. Go to the editor

*Sample data : 3, 5, 7, 23*

*Output :*

List : [3, 5, 7, 23]

Tuple : (3, 5, 7, 23)

[Click me to see the sample solution](#)

**7.** Write a Python program to accept a filename from the user and print the extension of that.

Go to the editor

*Sample filename : abc.java*

*Output : java*

[Click me to see the sample solution](#)

**8.** Write a Python program to display the first and last colors from the following list. Go to the editor

color\_list = ["Red","Green","White", "Black"]

[Click me to see the sample solution](#)

**9.** Write a Python program to display the examination schedule. (extract the date from exam\_st\_date). Go to the editor

exam\_st\_date = (11, 12, 2014)

Sample Output : The examination will start from : 11 / 12 / 2014

[Click me to see the sample solution](#)

**10.** Write a Python program that accepts an integer (n) and computes the value of n+nn+nnn.

Go to the editor

*Sample value of n is 5*

*Expected Result : 615*

[Click me to see the sample solution](#)

**11.** Write a Python program to print the documents (syntax, description etc.) of Python built-in function(s).

*Sample function : abs()*

*Expected Result :*

`abs(number) -> number`

Return the absolute value of the argument.

[Click me to see the sample solution](#)

**12.** Write a Python program to print the calendar of a given month and year.

*Note : Use 'calendar' module.*

[Click me to see the sample solution](#)

**13.** Write a Python program to print the following 'here document'. Go to the editor

*Sample string :*

a string that you "don't" have to escape

This

is a ..... multi-line

heredoc string -----> example

[Click me to see the sample solution](#)

**14.** Write a Python program to calculate number of days between two dates.

*Sample dates : (2014, 7, 2), (2014, 7, 11)*

*Expected output : 9 days*

[Click me to see the sample solution](#)

**15.** Write a Python program to get the volume of a sphere with radius 6.

[Click me to see the sample solution](#)

**16.** Write a Python program to get the difference between a given number and 17, if the number is greater than 17 return double the absolute difference. Go to the editor

[Click me to see the sample solution](#)

**17.** Write a Python program to test whether a number is within 100 of 1000 or 2000. Go to the editor

[Click me to see the sample solution](#)

**18.** Write a Python program to calculate the sum of three given numbers, if the values are equal then return three times of their sum. Go to the editor

[Click me to see the sample solution](#)

**19.** Write a Python program to get a new string from a given string where "Is" has been added to the front. If the given string already begins with "Is" then return the string unchanged. Go to the editor

[Click me to see the sample solution](#)

**20.** Write a Python program to get a string which is n (non-negative integer) copies of a given string. Go to the editor

[Click me to see the sample solution](#)

**21.** Write a Python program to find whether a given number (accept from the user) is even or odd, print out an appropriate message to the user. Go to the editor

[Click me to see the sample solution](#)

**22.** Write a Python program to count the number 4 in a given list. Go to the editor

[Click me to see the sample solution](#)

**23.** Write a Python program to get the n (non-negative integer) copies of the first 2 characters of a given string. Return the n copies of the whole string if the length is less than 2. Go to the editor

[Click me to see the sample solution](#)

**24.** Write a Python program to test whether a passed letter is a vowel or not. Go to the editor

[Click me to see the sample solution](#)

**25.** Write a Python program to check whether a specified value is contained in a group of values. Go to the editor

*Test Data :*

3 -> [1, 5, 8, 3] : True

-1 -> [1, 5, 8, 3] : False

[Click me to see the sample solution](#)

**26.** Write a Python program to create a histogram from a given list of integers. Go to the editor

[Click me to see the sample solution](#)

**27.** Write a Python program to concatenate all elements in a list into a string and return it. Go to the editor

[Click me to see the sample solution](#)

- 28.** Write a Python program to print all even numbers from a given numbers list in the same order and stop the printing if any numbers that come after 237 in the sequence. Go to the editor  
*Sample numbers list:*

```
1 numbers = [
2 386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953,
3 399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687,
4 815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742,
5 958, 743, 527
6]
```

[Click me to see the sample solution](#)

- 29.** Write a Python program to print out a set containing all the colors from colorlist\_1 which are not present in color\_list\_2. Go to the editor

*\_Test Data :*

```
colorlist_1 = set(["White", "Black", "Red"])
color_list_2 = set(["Red", "Green"])
```

*\_Expected Output :*

```
{'Black', 'White'}
```

[Click me to see the sample solution](#)

- 30.** Write a Python program that will accept the base and height of a triangle and compute the area. Go to the editor

[Click me to see the sample solution](#)

- 31.** Write a Python program to compute the greatest common divisor (GCD) of two positive integers. Go to the editor

[Click me to see the sample solution](#)

- 32.** Write a Python program to get the least common multiple (LCM) of two positive integers.

[Go to the editor](#)

[Click me to see the sample solution](#)

- 33.** Write a Python program to sum of three given integers. However, if two values are equal sum will be zero. Go to the editor

[Click me to see the sample solution](#)

**34.** Write a Python program to sum of two given integers. However, if the sum is between 15 to 20 it will return 20. Go to the editor

[Click me to see the sample solution](#)

**35.** Write a Python program that will return true if the two given integer values are equal or their sum or difference is 5. Go to the editor

[Click me to see the sample solution](#)

**36.** Write a Python program to add two objects if both objects are an integer type. Go to the editor

[Click me to see the sample solution](#)

**37.** Write a Python program to display your details like name, age, address in three different lines. Go to the editor

[Click me to see the sample solution](#)

**38.** Write a Python program to solve  $(x + y) * (x + y)$ . Go to the editor

*Test Data :* x = 4, y = 3

*Expected Output :*  $(4 + 3)^2 = 49$

[Click me to see the sample solution](#)

**39.** Write a Python program to compute the future value of a specified principal amount, rate of interest, and a number of years. Go to the editor

*Test Data :* amt = 10000, int = 3.5, years = 7

*Expected Output :* 12722.79

[Click me to see the sample solution](#)

**40.** Write a Python program to compute the distance between the points (x1, y1) and (x2, y2).

Go to the editor

[Click me to see the sample solution](#)

**41.** Write a Python program to check whether a file exists. Go to the editor

[Click me to see the sample solution](#)

**42.** Write a Python program to determine if a Python shell is executing in 32bit or 64bit mode on OS. Go to the editor

[Click me to see the sample solution](#)

**43.** Write a Python program to get OS name, platform and release information. Go to the editor  
[Click me to see the sample solution](#)

**44.** Write a Python program to locate Python site-packages. Go to the editor  
[Click me to see the sample solution](#)

**45.** Write a python program to call an external command in Python. Go to the editor  
[Click me to see the sample solution](#)

**46.** Write a python program to get the path and name of the file that is currently executing. Go to the editor  
[Click me to see the sample solution](#)

**47.** Write a Python program to find out the number of CPUs using. Go to the editor  
[Click me to see the sample solution](#)

**48.** Write a Python program to parse a string to Float or Integer. Go to the editor  
[Click me to see the sample solution](#)

**49.** Write a Python program to list all files in a directory in Python. Go to the editor  
[Click me to see the sample solution](#)

**50.** Write a Python program to print without newline or space. Go to the editor  
[Click me to see the sample solution](#)

**51.** Write a Python program to determine profiling of Python programs. Go to the editor  
Note: A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.  
[Click me to see the sample solution](#)

**52.** Write a Python program to print to stderr. Go to the editor  
[Click me to see the sample solution](#)

**53.** Write a python program to access environment variables. Go to the editor  
[Click me to see the sample solution](#)

**54.** Write a Python program to get the current username Go to the editor  
[Click me to see the sample solution](#)

**55.** Write a Python to find local IP addresses using Python's stdlib Go to the editor

[Click me to see the sample solution](#)

**56.** Write a Python program to get height and width of the console window. Go to the editor

[Click me to see the sample solution](#)

**57.** Write a Python program to get execution time for a Python method. Go to the editor

[Click me to see the sample solution](#)

**58.** Write a Python program to sum of the first n positive integers. Go to the editor

[Click me to see the sample solution](#)

**59.** Write a Python program to convert height (in feet and inches) to centimeters. Go to the editor

[Click me to see the sample solution](#)

**60.** Write a Python program to calculate the hypotenuse of a right angled triangle. Go to the editor

[Click me to see the sample solution](#)

**61.** Write a Python program to convert the distance (in feet) to inches, yards, and miles. Go to the editor

[Click me to see the sample solution](#)

**62.** Write a Python program to convert all units of time into seconds. Go to the editor

[Click me to see the sample solution](#)

**63.** Write a Python program to get an absolute file path. Go to the editor

[Click me to see the sample solution](#)

**64.** Write a Python program to get file creation and modification date/times. Go to the editor

[Click me to see the sample solution](#)

**65.** Write a Python program to convert seconds to day, hour, minutes and seconds. Go to the editor

[Click me to see the sample solution](#)

**66.** Write a Python program to calculate body mass index. Go to the editor

[Click me to see the sample solution](#)

**67.** Write a Python program to convert pressure in kilopascals to pounds per square inch, a millimeter of mercury (mmHg) and atmosphere pressure. Go to the editor

[Click me to see the sample solution](#)

**68.** Write a Python program to calculate the sum of the digits in an integer. Go to the editor

[Click me to see the sample solution](#)

**69.** Write a Python program to sort three integers without using conditional statements and loops. Go to the editor

[Click me to see the sample solution](#)

**70.** Write a Python program to sort files by date. Go to the editor

[Click me to see the sample solution](#)

**71.** Write a Python program to get a directory listing, sorted by creation date. Go to the editor

[Click me to see the sample solution](#)

**72.** Write a Python program to get the details of math module. Go to the editor

[Click me to see the sample solution](#)

**73.** Write a Python program to calculate midpoints of a line. Go to the editor

[Click me to see the sample solution](#)

**74.** Write a Python program to hash a word. Go to the editor

[Click me to see the sample solution](#)

**75.** Write a Python program to get the copyright information and write Copyright information in Python code. Go to the editor

[Click me to see the sample solution](#)

**76.** Write a Python program to get the command-line arguments (name of the script, the number of arguments, arguments) passed to a script. Go to the editor

[Click me to see the sample solution](#)

**77.** Write a Python program to test whether the system is a big-endian platform or little-endian platform. Go to the editor

[Click me to see the sample solution](#)

**78.** Write a Python program to find the available built-in modules. Go to the editor

[Click me to see the sample solution](#)

**79.** Write a Python program to get the size of an object in bytes. Go to the editor

[Click me to see the sample solution](#)

**80.** Write a Python program to get the current value of the recursion limit. Go to the editor

[Click me to see the sample solution](#)

**81.** Write a Python program to concatenate N strings. Go to the editor

[Click me to see the sample solution](#)

**82.** Write a Python program to calculate the sum of all items of a container (tuple, list, set, dictionary). Go to the editor

[Click me to see the sample solution](#)

**83.** Write a Python program to test whether all numbers of a list is greater than a certain number. Go to the editor

[Click me to see the sample solution](#)

**84.** Write a Python program to count the number occurrence of a specific character in a string. Go to the editor

[Click me to see the sample solution](#)

**85.** Write a Python program to check whether a file path is a file or a directory. Go to the editor

[Click me to see the sample solution](#)

**86.** Write a Python program to get the ASCII value of a character. Go to the editor

[Click me to see the sample solution](#)

**87.** Write a Python program to get the size of a file. Go to the editor

[Click me to see the sample solution](#)

**88.** Given variables  $x=30$  and  $y=20$ , write a Python program to print "30+20=50". Go to the editor

[Click me to see the sample solution](#)

**89.** Write a Python program to perform an action if a condition is true. Go to the editor

Given a variable name, if the value is 1, display the string "First day of a Month!" and do nothing

if the value is not equal.

[Click me to see the sample solution](#)

**90.** Write a Python program to create a copy of its own source code. Go to the editor

[Click me to see the sample solution](#)

**91.** Write a Python program to swap two variables. Go to the editor

[Click me to see the sample solution](#)

**92.** Write a Python program to define a string containing special characters in various forms.

[Go to the editor](#)

[Click me to see the sample solution](#)

**93.** Write a Python program to get the Identity, Type, and Value of an object. Go to the editor

[Click me to see the sample solution](#)

**94.** Write a Python program to convert a byte string to a list of integers. Go to the editor

[Click me to see the sample solution](#)

**95.** Write a Python program to check whether a string is numeric. Go to the editor

[Click me to see the sample solution](#)

**96.** Write a Python program to print the current call stack. Go to the editor

[Click me to see the sample solution](#)

**97.** Write a Python program to list the special variables used within the language. Go to the editor

[Click me to see the sample solution](#)

**98.** Write a Python program to get the system time. Go to the editor

Note : The system time is important for debugging, network information, random number seeds, or something as simple as program performance.

[Click me to see the sample solution](#)

**99.** Write a Python program to clear the screen or terminal. Go to the editor

[Click me to see the sample solution](#)

**100.** Write a Python program to get the name of the host on which the routine is running. Go to the editor

[Click me to see the sample solution](#)

**101.** Write a Python program to access and print a URL's content to the console. Go to the editor

[Click me to see the sample solution](#)

**102.** Write a Python program to get system command output. Go to the editor

[Click me to see the sample solution](#)

**103.** Write a Python program to extract the filename from a given path. Go to the editor

[Click me to see the sample solution](#)

**104.** Write a Python program to get the effective group id, effective user id, real group id, a list of supplemental group ids associated with the current process. Go to the editor

Note: Availability: Unix.

[Click me to see the sample solution](#)

**105.** Write a Python program to get the users environment. Go to the editor

[Click me to see the sample solution](#)

**106.** Write a Python program to divide a path on the extension separator. Go to the editor

[Click me to see the sample solution](#)

**107.** Write a Python program to retrieve file properties. Go to the editor

[Click me to see the sample solution](#)

**108.** Write a Python program to find path refers to a file or directory when you encounter a path name. Go to the editor

[Click me to see the sample solution](#)

**109.** Write a Python program to check if a number is positive, negative or zero. Go to the editor

[Click me to see the sample solution](#)

**110.** Write a Python program to get numbers divisible by fifteen from a list using an anonymous function. Go to the editor

[Click me to see the sample solution](#)

**111.** Write a Python program to make file lists from current directory using a wildcard. Go to the editor

[Click me to see the sample solution](#)

**112.** Write a Python program to remove the first item from a specified list. Go to the editor

[Click me to see the sample solution](#)

**113.** Write a Python program to input a number, if it is not a number generates an error message. Go to the editor

[Click me to see the sample solution](#)

**114.** Write a Python program to filter the positive numbers from a list. Go to the editor

[Click me to see the sample solution](#)

**115.** Write a Python program to compute the product of a list of integers (without using for loop). Go to the editor

[Click me to see the sample solution](#)

**116.** Write a Python program to print Unicode characters. Go to the editor

[Click me to see the sample solution](#)

**117.** Write a Python program to prove that two string variables of same value point same memory location. Go to the editor

[Click me to see the sample solution](#)

**118.** Write a Python program to create a bytearray from a list. Go to the editor

[Click me to see the sample solution](#)

**119.** Write a Python program to round a floating-point number to specified number decimal places. Go to the editor

[Click me to see the sample solution](#)

**120.** Write a Python program to format a specified string limiting the length of a string. Go to the editor

[Click me to see the sample solution](#)

**121.** Write a Python program to determine whether variable is defined or not. Go to the editor

[Click me to see the sample solution](#)

**122.** Write a Python program to empty a variable without destroying it. Go to the editor

Sample data: n=20

```
d = {"x":200}
```

Expected Output : 0

```
{}
```

[Click me to see the sample solution](#)

**123.** Write a Python program to determine the largest and smallest integers, longs, floats. Go to the editor

[Click me to see the sample solution](#)

**124.** Write a Python program to check whether multiple variables have the same value. Go to the editor

[Click me to see the sample solution](#)

**125.** Write a Python program to sum of all counts in a collections. Go to the editor

[Click me to see the sample solution](#)

**126.** Write a Python program to get the actual module object for a given object. Go to the editor

[Click me to see the sample solution](#)

**127.** Write a Python program to check whether an integer fits in 64 bits. Go to the editor

[Click me to see the sample solution](#)

**128.** Write a Python program to check whether lowercase letters exist in a string. Go to the editor

[Click me to see the sample solution](#)

**129.** Write a Python program to add leading zeroes to a string. Go to the editor

[Click me to see the sample solution](#)

**130.** Write a Python program to use double quotes to display strings. Go to the editor

[Click me to see the sample solution](#)

**131.** Write a Python program to split a variable length string into variables. Go to the editor

[Click me to see the sample solution](#)

**132.** Write a Python program to list home directory without absolute path. Go to the editor  
[Click me to see the sample solution](#)

**133.** Write a Python program to calculate the time runs (difference between start and current time) of a program. Go to the editor  
[Click me to see the sample solution](#)

**134.** Write a Python program to input two integers in a single line. Go to the editor  
[Click me to see the sample solution](#)

**135.** Write a Python program to print a variable without spaces between values. Go to the editor  
Sample value : x =30

Expected output : Value of x is "30"  
[Click me to see the sample solution](#)

**136.** Write a Python program to find files and skip directories of a given directory. Go to the editor  
[Click me to see the sample solution](#)

**137.** Write a Python program to extract single key-value pair of a dictionary in variables. Go to the editor  
[Click me to see the sample solution](#)

**138.** Write a Python program to convert true to 1 and false to 0. Go to the editor  
[Click me to see the sample solution](#)

**139.** Write a Python program to valid a IP address. Go to the editor  
[Click me to see the sample solution](#)

**140.** Write a Python program to convert an integer to binary keep leading zeros. Go to the editor  
Sample data : x=12  
Expected output : 00001100  
0000001100  
[Click me to see the sample solution](#)

**141.** Write a python program to convert decimal to hexadecimal. Go to the editor  
Sample decimal number: 30, 4  
Expected output: 1e, 04  
[Click me to see the sample solution](#)

**142.** Write a Python program to find the operating system name, platform and platform release date. Go to the editor

Operating system name:

posix

Platform name:

Linux

Platform release:

4.4.0-47-generic

[Click me to see the sample solution](#)

**143.** Write a Python program to determine if the python shell is executing in 32bit or 64bit mode on operating system. Go to the editor

[Click me to see the sample solution](#)

**144.** Write a Python program to check whether variable is integer or string. Go to the editor

[Click me to see the sample solution](#)

**145.** Write a Python program to test if a variable is a list or tuple or a set. Go to the editor

[Click me to see the sample solution](#)

**146.** Write a Python program to find the location of Python module sources. Go to the editor

[Click me to see the sample solution](#)

**147.** Write a Python function to check whether a number is divisible by another number. Accept two integers values form the user. Go to the editor

[Click me to see the sample solution](#)

**148.** Write a Python function to find the maximum and minimum numbers from a sequence of numbers. Go to the editor

Note: Do not use built-in functions.

[Click me to see the sample solution](#)

**149.** Write a Python function that takes a positive integer and returns the sum of the cube of all the positive integers smaller than the specified number. Go to the editor

[Click me to see the sample solution](#)

**150.** Write a Python function to check whether a distinct pair of numbers whose product is odd present in a sequence of integer values. Go to the editor

[Click me to see the sample solution](#)

## Direct

```
1 import math
2
3
4 def say_hi(name):
5 """<---- Multi-Line Comments and Docstrings
6 This is where you put your content for help() to inform the user
7 about what your function does and how to use it
8 """
9 print(f"Hello {name}!")
10
11
12 print(say_hi("Bryan")) # Should get the print inside the function, then
13 # Boolean Values
14 # Work the same as in JS, except they are title case: True and False
15 a = True
16 b = False
17 # Logical Operators
18 # ! = not, || = or, && = and
19 print(True and True)
20 print(True and not True)
21 print(True or True)
22 # Truthiness - Everything is True except...
23 # False - None, False, '', [], (), set(), range(0)
24 # Number Values
25 # Integers are numbers without a floating decimal point
26 print(type(3)) # type returns the type of whatever argument you pass in
27 # Floating Point values are numbers with a floating decimal point
28 print(type(3.5))
29 # Type Casting
30 # You can convert between ints and floats (along with other types...)
31 print(float(3)) # If you convert a float to an int, it will truncate the
32 print(int(4.5))
33 print(type(str(3)))
34 # Python does not automatically convert types like JS
35 # print(17.0 + ' heyooo ' + 17) # TypeError
36 # Arithmetic Operators
37 # ** - exponent (comparable to Math.pow(num, pow))
38 # // - integer division
39 # There is no ++ or -- in Python
40 # String Values
41 # We can use single quotes, double quotes, or f'' for string formats
42 # We can use triple single quotes for multiline strings
43 print(
44 """This here's a story
45 All about how
46 My life got twist
47 Turned upside down
48 """
```

```
49)
50 # Three double quotes can also be used, but we typically reserve these
51 # multi-line comments and function docstrings (refer to lines 6-9)(Nice)
52 # We use len() to get the length of something
53 print(len("Bryan G")) # 7 characters
54 print(len(["hey", "ho", "hey", "hey", "ho"])) # 5 list items
55 print(len({1, 2, 3, 4, 5, 6, 7, 9})) # 8 set items
56 # We can index into strings, list, etc..self.
57 name = "Bryan"
58 for i in range(len(name)):
59 print(name[i]) # B, r, y, a, n
60 # We can index starting from the end as well, with negatives
61 occupation = "Full Stack Software Engineer"
62 print(occupation[-3]) # e
63 # We can also get ranges in the index with the [start:stop:step] syntax
64 print(occupation[0:4:1]) # step and stop are optional, stop is exclusiv
65 print(occupation[::-4]) # beginning to end, every 4th letter
66 print(occupation[4:14:2]) # Let's get weird with it!
67 # NOTE: Indexing out of range will give you an IndexError
68 # We can also get the index og things with the .index() method, similar
69 print(occupation.index("Stack"))
70 print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
71 # We can count how many times a substring/item appears in something as w
72 print(occupation.count("S"))
73 print(
74 """Now this here's a story all about how
75 My life got twist turned upside down
76 I forget the rest but the the potato
77 smells like the potato""".count(
78 "the"
79)
80)
81 # We concatenate the same as Javascript, but we can also multiply strings
82 print("dog " + "show")
83 print("ha" * 10)
84 # We can use format for a multitude of things, from spaces to decimal p
85 first_name = "Bryan"
86 last_name = "Guner"
87 print("Your name is {0} {1}".format(first_name, last_name))
88 # Useful String Methods
89 print("Hello".upper()) # HELLO
90 print("Hello".lower()) # hello
91 print("HELLO".islower()) # False
92 print("HELLO".isupper()) # True
93 print("Hello".startswith("he")) # False
94 print("Hello".endswith("lo")) # True
95 print("Hello There".split()) # [Hello, There]
96 print("hello1".isalpha()) # False, must consist only of letters
97 print("hello1".isalnum()) # True, must consist of only letters and numbe
98 print("3215235123".isdecimal()) # True, must be all numbers
99 # True, must consist of only spaces/tabs/newlines
100 print("\n ".isspace())
101 # False, index 0 must be upper case and the rest lower
```

```
102 print("Bryan Guner".istitle())
103 print("Michael Lee".istitle()) # True!
104 # Duck Typing - If it walks like a duck, and talks like a duck, it must
105 # Assignment - All like JS, but there are no special keywords like let or const
106 a = 3
107 b = a
108 c = "heyoo"
109 b = ["reassignment", "is", "fine", "G!"]
110 # Comparison Operators - Python uses the same equality operators as JS,
111 # < - Less than
112 # > - Greater than
113 # <= - Less than or Equal
114 # >= - Greater than or Equal
115 # == - Equal to
116 # != - Not equal to
117 # is - Refers to exact same memory location
118 # not - !
119 # Precedence - Negative Signs(not) are applied first(part of each number)
120 # - Multiplication and Division(and) happen next
121 # - Addition and Subtraction(or) are the last step
122 # NOTE: Be careful when using not along with ==
123 print(not a == b) # True
124 # print(a == not b) # Syntax Error
125 print(a == (not b)) # This fixes it. Answer: False
126 # Python does short-circuit evaluation
127 # Assignment Operators - Mostly the same as JS except Python has **=
128 # Flow Control Statements - if, while, for
129 # Note: Python smushes 'else if' into 'elif'!
130 if 10 < 1:
131 print("We don't get here")
132 elif 10 < 5:
133 print("Nor here...")
134 else:
135 print("Hey there!")
136 # Looping over a string
137 for c in "abcdefghijklmnopqrstuvwxyz":
138 print(c)
139 # Looping over a range
140 for i in range(5):
141 print(i + 1)
142 # Looping over a list
143 lst = [1, 2, 3, 4]
144 for i in lst:
145 print(i)
146 # Looping over a dictionary
147 spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
148 for v in spam.values():
149 print(v)
150 # Loop over a list of tuples and destructuring the values
151 # Assuming spam.items returns a list of tuples each containing two items
152 for k, v in spam.items():
153 print(f"{k}: {v}")
154 # While loops as long as the condition is True
```

```
155 # - Exit loop early with break
156 # - Exit iteration early with continue
157 spam = 0
158 while True:
159 print("Sike That's the wrong Numba")
160 spam += 1
161 if spam < 5:
162 continue
163 break
164
165 # Functions - use def keyword to define a function in Python
166
167
168 def printCopyright():
169 print("Copyright 2021, Bgoonz")
170
171
172 # Lambdas are one liners! (Should be at least, you can use parenthesis)
173 def avg(num1, num2):
174 return print(num1 + num2)
175
176
177 avg(1, 2)
178 # Calling it with keyword arguments, order does not matter
179 avg(num2=20, num1=1252)
180 printCopyright()
181 # We can give parameters default arguments like JS
182
183
184 def greeting(name, saying="Hello"):
185 print(saying, name)
186
187
188 greeting("Mike") # Hello Mike
189 greeting("Bryan", saying="Hello there...")
190 # A common gotcha is using a mutable object for a default parameter
191 # All invocations of the function reference the same mutable object
192
193
194 def append_item(item_name, item_list=[]): # Will it obey and give us a
195 item_list.append(item_name)
196 return item_list
197
198
199 # Uses same item list unless otherwise stated which is counterintuitive
200 print	append_item("notebook"))
201 print	append_item("notebook"))
202 print	append_item("notebook", []))
203 # Errors - Unlike JS, if we pass the incorrect amount of arguments to a
204 # it will throw an error
205 # avg(1) # TypeError
206 # avg(1, 2, 2) # TypeError
207 # ----- DAY 2 -----
```

```

208 # Functions - * to get rest of position arguments as tuple
209 # - ** to get rest of keyword arguments as a dictionary
210 # Variable Length positional arguments
211
212
213 def add(a, b, *args):
214 # args is a tuple of the rest of the arguments
215 total = a + b
216 for n in args:
217 total += n
218 return total
219
220
221 print(add(1, 2)) # args is None, returns 3
222 print(add(1, 2, 3, 4, 5, 6)) # args is (3, 4, 5, 6), returns 21
223 # Variable Length Keyword Arguments
224
225
226 def print_names_and_countries(greeting, **kwargs):
227 # kwargs is a dictionary of the rest of the keyword arguments
228 for k, v in kwargs.items():
229 print(greeting, k, "from", v)
230
231
232 print_names_and_countries(
233 "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
234)
235 # We can combine all of these together
236
237
238 def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
239 pass
240
241
242 # Lists are mutable arrays
243 empty_list = []
244 roomates = ["Beau", "Delynn"]
245 # List built-in function makes a list too
246 specials = list()
247 # We can use 'in' to test if something is in the list, like 'includes'
248 print(1 in [1, 2, 4]) # True
249 print(2 in [1, 3, 5]) # False
250 # Dictionaries - Similar to JS POJO's or Map, containing key value pairs
251 a = {"one": 1, "two": 2, "three": 3}
252 b = dict(one=1, two=2, three=3)
253 # Can use 'in' on dictionaries too (for keys)
254 print("one" in a) # True
255 print(3 in b) # False
256 # Sets - Just like JS, unordered collection of distinct objects
257 bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}
258 # bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
259 school_bag = set(
260 ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"]

```

```
261)
262 print(school_bag)
263 print(bedroom)
264 # We can use 'in' on sets as well
265 print(1 in {1, 2, 3}) # True
266 print(4 in {1, 3, 5}) # False
267 # Tuples are immutable lists of items
268 time_blocks = ("AM", "PM")
269 colors = "red", "green", "blue" # Parenthesis not needed but encouraged
270 # The tuple built-in function can be used to convert things to tuples
271 print(tuple("abc"))
272 print(tuple([1, 2, 3]))
273 # 'in' may be used on tuples as well
274 print(1 in (1, 2, 3)) # True
275 print(5 in (1, 4, 3)) # False
276 # Ranges are immutable lists of numbers, often used with for loops
277 # - start - default: 0, first number in sequence
278 # - stop - required, next number past last number in sequence
279 # - step - default: 1, difference between each number in sequence
280 range1 = range(5) # [0,1,2,3,4]
281 range2 = range(1, 5) # [1,2,3,4]
282 range3 = range(0, 25, 5) # [0,5,10,15,20]
283 range4 = range(0) # []
284 for i in range1:
285 print(i)
286 # Built-in functions:
287 # Filter
288
289
290 def isOdd(num):
291 return num % 2 == 1
292
293
294 filtered = filter(isOdd, [1, 2, 3, 4])
295 print(list(filtered))
296 for num in filtered:
297 print(f"first way: {num}")
298 print("--" * 20)
299 [print(f"list comprehension: {i}")
300 for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
301 # Map
302
303
304 def toUpper(str):
305 return str.upper()
306
307
308 upperCased = map(toUpper, ["a", "b", "c", "d"])
309 print(list(upperCased))
310 # Sorted
311 sorted_items = sorted(["john", "tom", "sonny", "Mike"])
312 print(list(sorted_items)) # Notice uppercase comes before lowercase
313 # Using a key function to control the sorting and make it case insensitive
```

```
314 sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
315 print(sorted_items)
316 # You can also reverse the sort
317 sorted_items = sorted(["john", "tom", "sonny", "Mike"],
318 key=str.lower, reverse=True)
319 print(sorted_items)
320 # Enumerate creates a tuple with an index for what you're enumerating
321 quarters = ["First", "Second", "Third", "Fourth"]
322 print(list(enumerate(quarters)))
323 print(list(enumerate(quarters, start=1)))
324 # Zip takes list and combines them as key value pairs, or really however
325 keys = ("Name", "Email")
326 values = ("Buster", "cheetoh@johhnydepp.com")
327 zipped = zip(keys, values)
328 print(list(zipped))
329 # You can zip more than 2
330 x_coords = [0, 1, 2, 3, 4]
331 y_coords = [4, 6, 10, 9, 10]
332 z_coords = [20, 10, 5, 9, 1]
333 coords = zip(x_coords, y_coords, z_coords)
334 print(list(coords))
335 # Len reports the length of strings along with list and any other object
336 # doing this to save myself some typing
337
338
339 def print_len(item):
340 return print(len(item))
341
342
343 print_len("Mike")
344 print_len([1, 5, 2, 10, 3, 10])
345 print_len({1, 5, 10, 9, 10}) # 4 because there is a duplicate here (10)
346 print_len((1, 4, 10, 9, 20))
347 # Max will return the max number in a given scenario
348 print(max(1, 2, 35, 1012, 1))
349 # Min
350 print(min(1, 5, 2, 10))
351 print(min([1, 4, 7, 10]))
352 # Sum
353 print(sum([1, 2, 4]))
354 # Any
355 print(any([True, False, False]))
356 print(any([False, False, False]))
357 # All
358 print(all([True, True, False]))
359 print(all([True, True, True]))
360 # Dir returns all the attributes of an object including it's methods and
361 user = {"Name": "Bob", "Email": "bob@bob.com"}
362 print(dir(user))
363 # Importing packages and modules
364 # - Module - A Python code in a file or directory
365 # - Package - A module which is a directory containing an __init__.py
366 # - Submodule - A module which is contained within a package
```

```

367 # - Name - An exported function, class, or variable in a module
368 # Unlike JS, modules export ALL names contained within them without any
369 # Assuming we have the following package with four submodules
370 # math
371 # | __init__.py
372 # | addition.py
373 # | subtraction.py
374 # | multiplication.py
375 # | division.py
376 # If we peek into the addition.py file we see there's an add function
377 # addition.py
378 # We can import 'add' from other places because it's a 'name' and is au-
379
380
381 # def add(num1, num2):
382 # return num1 + num2
383
384
385 # Notice the . syntax because this package can import it's own submodule
386 # Our __init__.py has the following files
387 # This imports the 'add' function
388 # And now it's also re-exported in here as well
389 # from .addition import add
390 # These import and re-export the rest of the functions from the submodule
391 # from .subtraction import subtract
392 # from .division import divide
393 # from .multiplication import multiply
394 # So if we have a script.py and want to import add, we could do it many
395 # This will load and execute the 'math/__init__.py' file and give
396 # us an object with the exported names in 'math/__init__.py'
397 # print(math.add(1,2))
398 # This imports JUST the add from 'math/__init__.py'
399 # from math import add
400 # print(add(1, 2))
401 # This skips importing from 'math/__init__.py' (although it still runs)
402 # and imports directly from the addition.py file
403 # from math.addition import add
404 # This imports all the functions individually from 'math/__init__.py'
405 # from math import add, subtract, multiply, divide
406 # print(add(1, 2))
407 # print(subtract(2, 1))
408 # This imports 'add' renames it to 'add_some_numbers'
409 # from math import add as add_some_numbers
410 # ----- DAY 3 -----
411 # Classes, Methods, and Properties
412
413
414 class AngryBird:
415 # Slots optimize property access and memory usage and prevent you
416 # from arbitrarily assigning new properties to the instance
417 __slots__ = ["_x", "_y"]
418 # Constructor
419

```

```
420 def __init__(self, x=0, y=0):
421 # Doc String
422 """
423 Construct a new AngryBird by setting it's position to (0, 0)
424 """
425 # Instance Variables
426 self._x = x
427 self._y = y
428
429 # Instance Method
430
431 def move_up_by(self, delta):
432 self._y += delta
433
434 # Getter
435
436 @property
437 def x(self):
438 return self._x
439
440 # Setter
441
442 @x.setter
443 def x(self, value):
444 if value < 0:
445 value = 0
446 self._x = value
447
448 @property
449 def y(self):
450 return self._y
451
452 @y.setter
453 def y(self, value):
454 self._y = value
455
456 # Dunder Repr... called by 'print'
457
458 def __repr__(self):
459 return f"<AngryBird ({self._x}, {self._y})>"
460
461
462 # JS to Python Classes cheat table
463 # JS Python
464 # constructor() def __init__(self):
465 # super() super().__init__()
466 # this.property self.property
467 # this.method self.method()
468 # method(arg1, arg2){} def method(self, arg1, ...)
469 # get someProperty(){} @property
470 # set someProperty(){} @someProperty.setter
471 # List Comprehensions are a way to transform a list from one format to another
472 # - Pythonic Alternative to using map or filter
```

```
473 # - Syntax of a list comprehension
474 # - new_list = [value loop condition]
475 # Using a for loop
476 squares = []
477 for i in range(10):
478 squares.append(i ** 2)
479 print(squares)
480 # value = i ** 2
481 # loop = for i in range(10)
482 squares = [i ** 2 for i in range(10)]
483 print(list(squares))
484 sentence = "the rocket came back from mars"
485 vowels = [character for character in sentence if character in "aeiou"]
486 print(vowels)
487 # You can also use them on dictionaries. We can use the items() method
488 # for the dictionary to loop through it getting the keys and values out
489 person = {"name": "Corina", "age": 32, "height": 1.4}
490 # This loops through and capitalizes the first letter of all keys
491 newPerson = {key.title(): value for key, value in person.items()}
492 print(list(newPerson.items()))
```

## Gist

## Repl



StripedInternationalMotion

<https://replit.com/@bgoonz/StripedInternationalMotion#main.py>

# **Untitled**

# Generate a graph

Difficulty Level : Medium

- Last Updated : 28 Jun, 2021

## Prerequisite – Graphs

To draw graph using in built libraries – Graph plotting in Python

In this article, we will see how to implement graph in python using dictionary data structure in python.

The keys of the dictionary used are the nodes of our graph and the corresponding values are lists with each nodes, which are connecting by an edge.

This simple graph has six nodes (a-f) and five arcs:

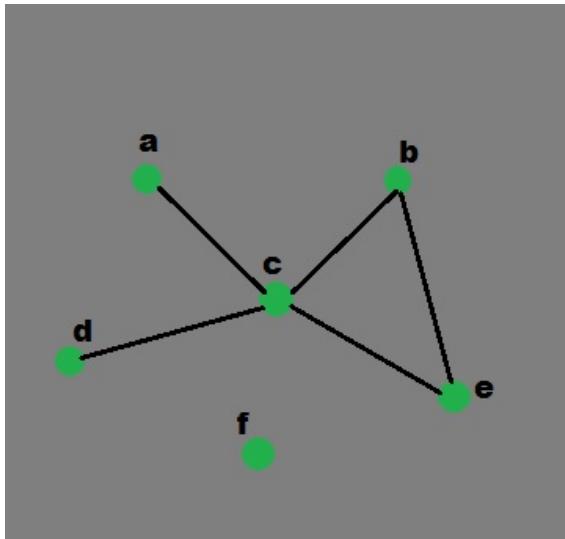
```
1 a -> c
2 b -> c
3 b -> e
4 c -> a
5 c -> b
6 c -> d
7 c -> e
8 d -> c
9 e -> c
10 e -> b
11
12
```

It can be represented by the following Python data structure. This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node.

```
1 graph = { "a" : ["c"],
2 "b" : ["c", "e"],
3 "c" : ["a", "b", "d", "e"],
4 "d" : ["c"],
```

```
5 "e" : ["c", "b"],
6 "f" : []
7 }
8
9
```

**Graphical representation of above example:**



**defaultdict:** Usually, a Python dictionary throws a `KeyError` if you try to get an item with a key that is not currently in the dictionary. `defaultdict` allows that if a key is not found in the dictionary, then instead of a `KeyError` being thrown, a new entry is created. The type of this new entry is given by the argument of `defaultdict`.

**Python Function to generate graph:**

```
1 # definition of function
2 def generate_edges(graph):
3 edges = []
4
5 # for each node in graph
6 for node in graph:
7
8 # for each neighbour node of a single node
9 for neighbour in graph[node]:
10 # if edge exists then append
11 edges.append((node, neighbour))
```

```
12 return edges
13
14
```

Recommended: Please try your approach on **{IDE}** first, before moving on to the solution.

---

## Python

 blank

<https://replit.com/@bgoonz/blank>

---

Output:

```
1 [('a', 'c'), ('c', 'd'), ('c', 'e'), ('c', 'a'), ('c', 'b'),
2 ('b', 'c'), ('b', 'e'), ('e', 'b'), ('e', 'c'), ('d', 'c')]
3
4
```

As we have taken example of undirected graph, so we have print same edge twice say as ('a','c') and ('c','a'). We can overcome this with use of directed graph.

Below are some more programs on graphs in python:

### 1. To generate the path from one node to the other node:

Using Python dictionary, we can find the path from one node to the other in a Graph. The

idea is similar to DFS in graphs.

In the function, initially, the path is an empty list. In the starting, if the start node matches with the end node, the function will return the path. Otherwise the code goes forward and hits all the values of the starting node and searches for the path using recursion.

---

## Python



blank-1

<https://replit.com/@bgoonz/blank-1#main.py>

1. Output:

```
1 ['d', 'a', 'c']
2
3
4
```

1.

### 2. Program to generate all the possible paths from one node to the other.:

In the above discussed program, we generated the first possible path. Now, let us generate all the possible paths from the start node to the end node. The basic functioning works same as the functioning of the above code. The place where the difference comes is instead of instantly returning the first path, it saves that path in a list named as 'paths' in the example given below. Finally, after iterating over all the possible ways, it returns the list of paths. If there is no path from the starting node to the ending node, it returns None.

---

## Python



possible-paths.py

<https://gist.github.com/bgoonz/b46192bc87d3cac28bdcf392afbe842d>

## Output:

```
1 [[['d', 'a', 'c'], ['d', 'a', 'c']]
2
3
4
```

1.

### 2. Program to generate the shortest path.:

To get to the shortest from all the paths, we use a little different approach as shown below. In this, as we get the path from the start node to the end node, we compare the length of the path with a variable named as shortest which is initialized with the None value. If the length of generated path is less than the length of shortest, if shortest is not None, the newly generated path is set as the value of shortest. Again, if there is no path, it returns None

- Python

```
Python program to generate shortest path graph
= {'a': ['c'], 'b': ['d'], 'c': ['e'], 'd': ['a', 'd'], 'e': ['b', 'c']}
function to find the shortest pathdef
find_shortest_path(graph, start, end, path = []):
 path = path + [start]
 if start == end:
 return path
 shortest = None
 for node in graph[start]:
 if node not in path:
 newpath = find_shortest_path(graph, node, end, path)
 if not shortest or len(newpath) < len(shortest):
 shortest = newpath
 return shortest
Driver function call to print# the shortest
pathprint(find_shortest_path(graph,
'd', 'c'))
```

### 1. Output:

```
['d', 'a', 'c']
```

# All Along

# **Beginners Guide To Python**

My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library support, and optional...

---

## **Beginners Guide To Python**

**My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library support, and optional named parameters.**

| Bram Cohen

**Article on basic web development setup... it is geared towards web but VSCode is an incredibly versatile editor and this stack really could suit just about anyone working in the field of computer science.**



sys Variables		String Methods		Datetime Methods	
argv	Command line args	capitalize() *	lstrip()	today()	fromordinal(ordinal)
builtin_module_names	Linked C modules	center(width)	partition(sep)	now(timezoneinfo)	combine(date, time)
byteorder	Native byte order	count(sub, start, end)	replace(old, new)	utcnow()	strftime(date, format)
check_interval	Signal check frequency	decode()	rfind(sub, start, end)	fromtimestamp(timestamp)	utcfromtimestamp(timestamp)
exec_prefix	Root directory	encode()	rindex(sub, start, end)		
executable	Name of executable	endswith(sub)	rjust(width)		
exitfunc	Exit function name	expandtabs()	rpartition(sep)		
modules	Loaded modules	find(sub, start, end)	rsplit(sep)		
path	Search path	index(sub, start, end)	rstrip()		
platform	Current platform	isalnum() *	split(sep)		
stdin, stdout, stderr	File objects for I/O	isalpha() *	splines()		
version_info	Python version info	isdigit() *	startswith(sub)		
winver	Version number	islower() *	strip()		
<b>sys.argv for \$ python foo.py bar -c qux --h</b>		isspace() *	swapcase() *		
sys.argv[0]	foo.py	istitle() *	title() *		
sys.argv[1]	bar	isupper() *	translate(table)		
sys.argv[2]	-c	join()	upper() *		
sys.argv[3]	qux	ljust(width)	zfill(width)		
sys.argv[4]	--h	lower() *			
<b>Note</b>		Methods marked * are locale dependant for 8-bit strings.			
os Variables		List Methods		Time Methods	
altsep	Alternative sep	append(item)	pop(position)	replace()	utcoffset()
curdir	Current dir string	count(item)	remove(item)	isoformat()	dst()
defpath	Default search path	extend(list)	reverse()	__str__()	tzname()
devnull	Path of null device	index(item)	sort()	strftime(format)	
extsep	Extension separator	insert(position, item)			
linesep	Line separator				
name	Name of OS				
pardir	Parent dir string				
pathsep	Patch separator				
sep	Path separator				
<b>Note</b>					
Registered OS names: "posix", "nt", "mac", "os2", "ce", "java", "riscos"					
Class Special Methods		File Methods		Date Formatting (strftime and strptime)	
__new__(cls)	__lt__(self, other)	close()	readlines(size)	%a Abbreviated weekday (Sun)	
__init__(self, args)	__le__(self, other)	flush()	seek(offset)	%A Weekday (Sunday)	
__del__(self)	__gt__(self, other)	fileno()	tell()	%b Abbreviated month name (Jan)	
__repr__(self)	__ge__(self, other)	isatty()	truncate(size)	%B Month name (January)	
__str__(self)	__eq__(self, other)	next()	write(string)	%c Date and time	
__cmp__(self, other)	__ne__(self, other)	read(size)	writelines(list)	%d Day (leading zeros) (01 to 31)	
__index__(self)	__nonzero__(self)	readline(size)		%H 24 hour (leading zeros) (00 to 23)	
__hash__(self)				%I 12 hour (leading zeros) (01 to 12)	
__getattr__(self, name)				%j Day of year (001 to 366)	
__getattribute__(self, name)				%m Month (01 to 12)	
__setattr__(self, name, attr)				%M Minute (00 to 59)	
__delattr__(self, name)				%p AM or PM	
__call__(self, args, kwargs)				%S Second (00 to 61*)	
				%U Week number ^ (00 to 53)	
				%w Weekday ^ (0 to 6)	
				%W Week number ^ (00 to 53)	
				%x Date	
				%X Time	
				%y Year without century (00 to 99)	
				%Y Year (2008)	
				%Z Time zone (GMT)	
				%% A literal "%" character (%)	
<b>Indexes and Slices (of a=[0,1,2,3,4,5])</b>		1. Sunday as start of week. All days in a new year preceding the first Sunday are considered to be in week 0.		2. 0 is Sunday, 6 is Saturday.	
len(a)	6	a[0]	0	3. Monday as start of week. All days in a new year preceding the first Monday are considered to be in week 0.	
a[0]	0	a[5]	5	4. This is not a mistake. Range takes account of leap and double-leap seconds.	
a[-1]	5	a[-2]	4		
a[1:-1]	[1,2,3,4,5]	a[1:3]	[1,2]		
a[:5]	[0,1,2,3,4]	a[1:-1]	[1,2,3,4]		
a[:-2]	[0,1,2,3]	b=a[:]	Shallow copy of a		

Available free from [AddedBytes.com](http://AddedBytes.com)

## Python

- Python is an interpreted, high-level and general-purpose, dynamically typed programming language
- It is also Object oriented, modular oriented and a scripting language.
- In Python, everything is considered as an Object.
- A python file has an extension of .py
- Python follows Indentation to separate code blocks instead of flower brackets({}).
- We can run a python file by the following command in cmd(Windows) or shell(mac/linux).
- `python <filename.py>`

**By default, the python doesn't require any imports to run a python file.**

## Create and execute a program

1. Open up a terminal/cmd
  2. Create the program: nano/cat > nameProgram.py
  3. Write the program and save it
  4. python nameProgram.py
- 

## Basic Datatypes

### Basic Datatypes

Data Type	Description
int	Integer values [0, 1, -2, 3]
float	Floating point values [0.1, 4.532, -5.092]
char	Characters [a, b, @, !, `]
str	Strings [abc, AbC, A@B, sd!, `asa]
bool	Boolean Values [True, False]
char	Characters [a, b, @, !, `]
complex	Complex numbers [2+3j, 4-1j]

Data Type Description  
int Integer values [0, 1, -2, 3]  
float Floating point values [0.1, 4.532, -5.092]  
char Characters [a, b, @, !, `]  
str Strings [abc, AbC, A@B, sd!, `asa]  
bool Boolean Values [True, False]  
char Characters [a, b, @, !, `]  
complex Complex numbers [2+3j, 4-1j]

---

## Keywords

## Keywords

Keyword	Description
break	used to exit loop and used to exit
char	basic declaration of a type character
const	prefix declaration meaning variable can not be changed
continue	go to bottom of loop in for, while loops
class	to define a class
def	to define a function
elif	shortcut for (else if) used in else if ladder
else	executable statement, part of "if" structure
float	basic declaration of floating point
for	executable statement, for loop
from	executable statement, used to import only specific objects from a package
if	executable statement
import	to import modules
pass	keyword to specify nothing is happening in the codeblock, generally used in classes
return	executable statement with or without a value
while	executable statement, while loop

KeywordDescription  
breakused to exit loop and used to exit  
charbasic declaration of a type  
character  
constprefix declaration meaning variable can not be changed  
continuego to bottom of  
loop in for, while loops  
class to define a class  
defto define a function  
elifshortcut for (else if) used in else if ladder  
elseexecutable statement, part of "if" structure  
floatbasic declaration of floating  
point  
forexecutable statement, for loop  
fromexecutable statement, used to import only specific  
objects from a package  
ife

---

## Operators

## Operators

Operator	Description
( )	grouping parenthesis, function call, tuple declaration
[ ]	array indexing, also declaring lists etc.
!	relational not, complement, ! a yields true or false
~	bitwise not, ones complement, ~a
-	unary minus, - a
+	unary plus, + a
*	multiply, a * b
/	divide, a / b
%	modulo, a % b
+	add, a + b
-	subtract, a - b
<<	shift left, left operand is shifted left by right operand bits
>>	shift right, left operand is shifted right by right operand bits
<	less than, result is true or false, a < b
<=	less than or equal, result is true or false, a <= b

>	greater than, result is true or false, a > b
>=	greater than or equal, result is true or false, a >= b
==	equal, result is true or false, a == b
!=	not equal, result is true or false, a != b
&	bitwise and, a & b
^	bitwise exclusive or XOR, a ^ b
	bitwise or, a
&&, and	relational and, result is true or false, a < b && c >= d
, or	relational or, result is true or false, a < b    c >= d
=	store or assignment
+=	add and store
-=	subtract and store
*=	multiply and store
/=	divide and store
%=	modulo and store
<<=	shift left and store
>>=	shift right and store
&=	bitwise and and store
^=	bitwise exclusive or and store
=	bitwise or and store
,	separator as in ( y=x,z=++x )

## Basic Data Structures

### List

- List is a collection which is ordered and changeable. Allows duplicate members.
- Lists are created using square brackets:

```
thislist = ["apple", "banana", "cherry"]
```

- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0] , the second item has index [1] etc.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- To determine how many items a list has, use the len() function.

- A list can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
```

- It is also possible to use the list() constructor when creating a new list

```
thislist = list(("apple", "banana", "cherry")) # note the double round-bracket
```

## Tuple

- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.

```
thistuple = ("apple", "banana", "cherry")
```

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0] , the second item has index [1] etc.
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Since tuple are indexed, tuples can have items with the same value:
- Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
```

- To determine how many items a tuple has, use the len() function:

```
1 thistuple = ("apple", "banana", "cherry")
2 print(len(thistuple))
```

- To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
1 thistuple = ("apple",)
2 print(type(thistuple))
```

```
1 #NOT a tuple
2 thistuple = ("apple")
3 print(type(thistuple))
```

- It is also possible to use the tuple() constructor to make a tuple.

```
1 thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
2 print(thistuple)
```

## Set

- Set is a collection which is unordered and unindexed. No duplicate members.
- A set is a collection which is both unordered and unindexed.

```
thisset = {"apple", "banana", "cherry"}
```

- Set items are unordered, unchangeable, and do not allow duplicate values.
- Unordered means that the items in a set do not have a defined order.

- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Sets are unchangeable, meaning that we cannot change the items after the set has been created.
- Duplicate values will be ignored.
- To determine how many items a set has, use the `len()` method.

```
thisset = {"apple", "banana", "cherry"}
```

```
print(len(thisset))
```

- Set items can be of any data type:

```
1 set1 = {"apple", "banana", "cherry"}
2 set2 = {1, 5, 7, 9, 3}
3 set3 = {True, False, False}
4 set4 = {"abc", 34, True, 40, "male"}
```

- It is also possible to use the `set()` constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
```

## Dictionary

- Dictionary is a collection which is unordered and changeable. No duplicate members.
- Dictionaries are used to store data values in key:value pairs.
- Dictionaries are written with curly brackets, and have keys and values:

```
1 thisdict = {
2 "brand": "Ford",
3 "model": "Mustang",
4 "year": 1964
5 }
```

- Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

```
1 thisdict = {
2 "brand": "Ford",
3 "model": "Mustang",
4 "year": 1964
5 }
6 print(thisdict["brand"])
```

- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key.
- Duplicate values will overwrite existing values.
- To determine how many items a dictionary has, use the `len()` function.

```
print(len(thisdict))
```

- The values in dictionary items can be of any data type

```
1 thisdict = {
2 "brand": "Ford",
3 "electric": False,
4 "year": 1964,
5 "colors": ["red", "white", "blue"]
6 }
```

# Conditional branching

```
1 if condition:
2 pass
3 elif condition2:
4 pass
5 else:
6 pass
```

# Loops

Python has two primitive loop commands:

1. while loops
2. for loops

## While loop

- With the `while` loop we can execute a set of statements as long as a condition is true.
- Example: Print i as long as i is less than 6

```
1 i = 1
2 while i < 6:
3 print(i)
4 i += 1
```

- The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.
- With the `break` statement we can stop the loop even if the while condition is true
- With the `continue` statement we can stop the current iteration, and continue with the next.
- With the `else` statement we can run a block of code once when the condition no longer is true.

## For loop

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3 print(x)
```

- The for loop does not require an indexing variable to set beforehand.
- To loop through a set of code a specified number of times, we can use the range() function.
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3).
- The else keyword in a for loop specifies a block of code to be executed when the loop is finished.

A nested loop is a loop inside a loop.

- The “inner loop” will be executed one time for each iteration of the “outer loop”:

```
1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]
```

```
1 for x in adj:
2 for y in fruits:
3 print(x, y)
```

- for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
1 for x in [0, 1, 2]:
```

```
2 pass
```

## Function definition

```
1 def function_name():
2 return
```

## Function call

```
function_name()
```

- We need not to specify the return type of the function.
- Functions by default return `None`
- We can return any datatype.

## Python Syntax

Python syntax was made for readability, and easy editing. For example, the python language uses a `:` and indented code, while javascript and others generally use `{}` and indented code.

## First Program

Lets create a python 3 repl, and call it *Hello World*. Now you have a blank file called `main.py`. Now let us write our first line of code:

```
print('Hello world!')
```

*Brian Kernighan actually wrote the first "Hello, World!" program as part of the documentation for the BCPL programming language developed by Martin Richards.*

Now, press the run button, which obviously runs the code. If you are not using replit, this will not work. You should research how to run a file with your text editor.

---

## Command Line

If you look to your left at the console where hello world was just printed, you can see a `>`, `>>>`, or `$` depending on what you are using. After the prompt, try typing a line of code.

```
1 Python 3.6.1 (default, Jun 21 2017, 18:48:35)
2 [GCC 4.9.2] on linux
3 Type "help", "copyright", "credits" or "license" for more information.
4 > print('Testing command line')
5 Testing command line
6 > print('Are you sure this works?')
7 Are you sure this works?
8 >
```

The command line allows you to execute single lines of code at a time. It is often used when trying out a new function or method in the language.

---

## New: Comments!

Another cool thing that you can generally do with all languages, are comments. In python, a comment starts with a `#`. The computer ignores all text starting after the `#`.

```
Write some comments!
```

If you have a huge comment, do **not** comment all the 350 lines, just put `'''` before it, and `'''` at the end. Technically, this is not a comment but a string, but the computer still ignores it, so we will use it.

---

## New: Variables!

Unlike many other languages, there is no `var`, `let`, or `const` to declare a variable in python. You simply go `name = 'value'`.

Remember, there is a difference between integers and strings. *Remember: String = ""*. To convert between these two, you can put an int in a `str()` function, and a string in a `int()` function. There is also a less used one, called a float. Mainly, these are integers with decimals. Change them using the `float()` command.

[https://repl.it/@bgoonz/second-scr?  
lite=true&referrer=https%3A%2F%2Fbryanguner.medium.com](https://repl.it/@bgoonz/second-scr?lite=true&referrer=https%3A%2F%2Fbryanguner.medium.com)

```
1 x = 5
2 x = str(x)
3 b = '5'
4 b = int(b)
5 print('x = ', x, '; b = ', str(b), ';') # => x = 5; b = 5;
```

Instead of using the `,` in the print function, you can put a `+` to combine the variables and string.

---

## Operators

There are many operators in python:

- `+`
- `-`
- `/`

- \*

These operators are the same in most languages, and allow for addition, subtraction, division, and multiplication.

Now, we can look at a few more complicated ones:

## Python Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+X, -X, ~X	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
		Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, Identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

*simpleops.py*

```

1 x = 4
2 a = x + 1
3 a = x - 1
4 a = x * 2
5 a = x / 2

```

You should already know everything shown above, as it is similar to other languages. If you continue down, you will see more complicated ones.

*complexop.py*

```
1 a += 1
2 a -= 1
3 a *= 2
4 a /= 2
```

The ones above are to edit the current value of the variable.

Sorry to JS users, as there is no `i++;` or anything.

---

## Fun Fact:

**The python language was named after Monty Python.**

If you really want to know about the others, view Py Operators

---

## More Things With Strings

Like the title?

Anyways, a `'` and a `"` both indicate a string, but **do not combine them!**

*quotes.py*

```
1 x = 'hello' # Good
2 x = "hello" # Good
3 x = "hello' # ERRORRR!!!
```

*slicing.py*

---

# String Slicing

You can look at only certain parts of the string by slicing it, using `[num:num]` .

The first number stands for how far in you go from the front, and the second stands for how far in you go from the back.

```
1 x = 'Hello everybody!'
2 x[1] # 'e'
3 x[-1] # '!'
4 x[5] # ' '
5 x[1:] # 'ello everybody!'
6 x[:-1] # 'Hello everybod'
7 x[2:-3] # 'llo everyb'
```

## Methods and Functions

Here is a list of functions/methods we will go over:

- `.strip()`
- `len()`
- `.lower()`
- `.upper()`
- `.replace()`
- `.split()`

## New: `input()`

`input` is a function that gathers input entered from the user in the command line. It takes one optional parameter, which is the users prompt.

*inp.py*

```
1 print('Type something: ')
```

```
2 x = input()
3 print('Here is what you said: ', x)
```

If you wanted to make it smaller, and look neater to the user, you could do...

*inp2.py*

```
print('Here is what you said: ', input('Type something: '))
```

Running:

*inp.py*

```
1 Type something:
2 Hello World
3 Here is what you said: Hello World
```

*inp2.py*

```
1 Type something: Hello World
2 Here is what you said: Hello World
```

## New: Importing Modules

Python has created a lot of functions that are located in other .py files. You need to import these **modules** to gain access to them. You may wonder why python did this. The purpose of separate modules is to make python faster. Instead of storing millions and millions of functions, it only needs a few basic ones. To import a module, you must write

```
input <modulename> . Do not add the .py extension to the file name. In this example , we will be using a python created module named random.
```

*module.py*

```
import random
```

Now, I have access to all functions in the random.py file. To access a specific function in the module, you would do `<module>.<function>`. For example:

*module2.py*

```
1 import random
2 print(random.randint(3,5)) # Prints a random number between 3 and 5
```

*Pro Tip:*

*Do `from random import randint` to not have to do `random.randint()`, just `randint()`*  
*To import all functions from a module, you could do `from random import *`*

## New: Loops!

Loops allow you to repeat code over and over again. This is useful if you want to print Hi with a delay of one second 100 times.

### for Loop

The for loop goes through a list of variables, making a separate variable equal one of the list every time.

Let's say we wanted to create the example above.

*loop.py*

```
1 from time import sleep
2 for i in range(100):
3 print('Hello')
4 sleep(.3)
```

This will print Hello with a .3 second delay 100 times. This is just one way to use it, but it is usually used like this:

*loop2.py*

```
1 import time
2 for number in range(100):
3 print(number)
4 time.sleep(.1)
```

[https://storage.googleapis.com/replit/images/1539649280875\\_37d22e6d49e8e8fbc453631def345387.png](https://storage.googleapis.com/replit/images/1539649280875_37d22e6d49e8e8fbc453631def345387.png)

## while Loop

The while loop runs the code while something stays true. You would put `while <expression>`. Every time the loop runs, it evaluates if the expression is True. If it is, it runs the code, if not it continues outside of the loop. For example:

*while.py*

```
1 while True: # Runs forever
2 print('Hello World!')
```

Or you could do:

*while2.py*

```
1 import random
2 position = '<placeholder>'
3 while position != 1: # will run at least once
4 position = random.randint(1, 10)
5 print(position)
```

## New: if Statement

The if statement allows you to check if something is True. If so, it runs the code, if not, it continues on. It is kind of like a while loop, but it executes **only once**. An if statement is written:

*if.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('num is 3. Hooray!!!')
5 if num > 5:
6 print('Num is greater than 5')
7 if num == 12:
8 print('Num is 12, which means that there is a problem with the python lan
```

Now, you may think that it would be better if you could make it print only one message. Not as many that are True. You can do that with an `elif` statement:

*elif.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('Num is three, this is the only msg you will see.')
5 elif num > 2:
6 print('Num is not three, but is greater than 1')
```

Now, you may wonder how to run code if none work. Well, there is a simple statement called `else:`

*else.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('Num is three, this is the only msg you will see.'
```

```
5 elif num > 2:
6 print('Num is not three, but is greater than 1')
7 else:
8 print('No category')
```

## New: Functions ( def )

So far, you have only seen how to use functions other people have made. Let use the example that you want to print the a random number between 1 and 9, and print different text every time. It is quite tiring to type:

Characters: 389

*nofunc.py*

```
1 import random
2 print(random.randint(1, 9))
3 print('Wow that was interesting.')
4 print(random.randint(1, 9))
5 print('Look at the number above ^')
6 print(random.randint(1, 9))
7 print('All of these have been interesting numbers.')
8 print(random.randint(1, 9))
9 print("these random.randint's are getting annoying to type")
10 print(random.randint(1, 9))
11 print('Hi')
12 print(random.randint(1, 9))
13 print('j')
```

Now with functions, you can seriously lower the amount of characters:

Characters: 254

*functions.py*

```
1 import random
2 def r(t):
```

```
3 print(random.randint(1, 9))
4 print(t)
5 r('Wow that was interesting.')
6 r('Look at the number above ^')
7 r('All of these have been interesting numbers.')
8 r("these random.randint's are getting annoying to type")
9 r('Hi')
10 r('j')
```

## Project Based Learning:

The following is a modified version of a tutorial posted By: InvisibleOne

I would cite the original tutorial it's self but at the time of this writing I can no longer find it on his repl.it profile and so the only reference I have are my own notes from following the tutorial when I first found it.

### 1. Adventure Story

The first thing you need with an adventure story is a great storyline, something that is exciting and fun. The idea is, that at each pivotal point in the story, you give the player the opportunity to make a choice.

First things first, let's import the stuff that we need, like this:

```
1 import os #very useful for clearing the screen
2 import random
```

Now, we need some variables to hold some of the player data.

```
1 name = input("Name Please: ") #We'll use this to get the name from the user
2 nickname = input("Nickname: ")
```

Ok, now we have the player's name and nickname, let's welcome them to the game

```
print("Hello and welcome " + name)
```

Now for the story. The most important part of all stories is the introduction, so let's print our introduction

```
1 print("Long ago, there was a magical meal known as Summuh and Spich Atip") #We
2 print("It was said that this meal had the power to save lives, restore peace,
```

Now, we'll give the player their first choice

```
1 print("After hiking through the wastelands for a long time, you come to a mass")
2 choice1 = input("[1] Take the bridge [2] Try and jump over")
3 #Now we check to see what the player chose
4 If choice1 == '1':
5 print("You slowly walk across the bridge, it creaks ominously, then suddenly")
6 #The player lost, so now we'll boot them out of the program with the exit co
7 exit()
8 #Then we check to see if they made the other choice, we can do with with else
9 elif choice1 == '2':
10 print("You make the jump! You see a feather hit the bridge, the weight break
11 #Now we can continue the story
12 print("A few more hours of travel and you come to the unclimbable mountain.")
13 choice2 == input("[1] Give up [2] Try and climb the mountain")
14 if choice2 == '1':
15 print("You gave up and lost...")
16 #now we exit them again
17 exit()
18 elif choice2 == '1':
19 print("you continue up the mountain. Climbing is hard, but finally you reach
20 print("Old Man: Hey " + nickname)
21 print("You: How do you know my name!?!")
22 print("Old Man: Because you have a name tag on...")
23 print("You: Oh, well, were is the Summuh and Spich Atip?")
24 print("Old Man: Summuh and Spich Atip? You must mean the Pita Chips and Humr
25 print("You: Pita...chips...humus, what power do those have?")
26 print("Old Man: Pretty simple kid, their organic...")
27 #Now let's clear the screen
28 os.system('clear')
29 print("YOU WON!!!!")
```

---

There you have it, a pretty simple choose your own ending story. You can make it as complex or uncomplex as you like.

---

## 2. TEXT ENCODER

Ever make secret messages as a kid? I used to. Anyways, here's the way you can make a program to encode messages! It's pretty simple. First things first, let's get the message the user wants to encode, we'll use `input()` for that:

```
message = input("Message you would like encoded: ")
```

Now we need to split that string into a list of characters, this part is a bit more complicated.

```
1 #We'll make a function, so we can use it later
2 def split(x):
3 return [char for char in x]
4 #now we'll call this function with our text
5 l_message = message.lower() #This way we can lower any of their input
6 encode = split(l_message)
```

Now we need to convert the characters into code, well do this with a for loop:

```
1 out = []
2 for x in encode:
3 if x == 'a':
4 out.append('1')
5 elif x == 'b':
6 out.append('2')
7 #And we'll continue on though this with each letter of the alphabet
```

---

Once we've encoded the text, we'll print it back for the user

---

```
1 x = ' '.join(out)
2 #this will turn out into a string that we can print
3 print(x)
```

And if you want to decode something, it is this same process but in reverse!

---

### 3. Guess my Number

Number guessing games are fun and pretty simple, all you need are a few loops. To start, we need to import random.

```
import random
```

That is pretty simple. Now we'll make a list with the numbers we want available for the game

```
num_list = [1,2,3,4,5,6,7,8,9,10]
```

Next, we get a random number from the list

```
num = random.choice(num_list)
```

Now, we need to ask the user for input, we'll do this with a while loop

```
1 while True:
2 # We could use guess = input("What do you think my number is? "), but that
3 # would have required us to use str()
4 # Next, we'll check if that number is equal to the number we picked
5 if guess == num:
6 break #this will remove us from the loop, so we can display the win message
7 else:
```

```
8 print("Nope, that isn't it")
9 #outside our loop, we'll have the win message that is displayed if the player
10 print("You won!")
```

Have fun with this!

---

## 4. Notes

Here is a more advanced project, but still pretty easy. This will be using a txt file to save some notes. The first thing we need to do is to create a txt file in your repl, name it 'notes.txt'

Now, to open a file in python we use `open('filename', type)` The type can be 'r' for read, or 'w' for write. There is another option, but we won't be using that here. Now, the first thing we are going to do is get what the user would like to save:

```
message = input("What would you like to save?")
```

Now we'll open our file and save that text

```
1 o = open('notes.txt', 'w')
2 o.write(message)
3 #this next part is very important, you need to always remember to close your
4 o.close()
```

There we go, now the information is in the file. Next, we'll retrieve it

```
1 read = open('notes.txt', 'r')
2 out = read.read()
3 # now we need to close the file
4 read.close()
5 # and now print what we read
6 print(out)
```

There we go, that's how you can open files and close files with python

---

## 5. Random Dare Generator

Who doesn't love a good dare? Here is a program that can generate random dares. The first thing we'll need to do is as always, import random. Then we'll make some lists of dares

```
1 import random
2 list1 = ['jump on', 'sit on', 'rick roll on', 'stop on', 'swing on']
3 list2 = ['your cat', 'your neighbor', 'a dog', 'a tree', 'a house']
4 list3 = ['your mom', 'your best friend', 'your dad', 'your teacher']
5 #now we'll generate a dare
6 while True:
7 if input() == '': #this will trigger if they hit enter
8 print("I dare you to " + random.choice(list1) + ' ' + random.choice(list2))
```



**PythonPracticeGists-1**

<https://replit.com/@bgoonz/PythonPracticeGists-1>

# Exercises



PythonPracticeGists

<https://replit.com/@bgoonz/Python-Practice-Gists>

\*\*\*\*

**Long list of small examples at bottom of page...**

\*\*\*\*

1. Write a Python program to print the following string in a specific format (see the output). Go to the editor

*Sample String :* "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are" *Output :*

```
1 Twinkle, twinkle, little star,
2 How I wonder what you are!
3 Up above the world so high,
4 Like a diamond in the sky.
5 Twinkle, twinkle, little star,
6 How I wonder what you are
```

Click me to see the sample solution

2. Write a Python program to get the Python version you are using. Go to the editor

Click me to see the sample solution

3. Write a Python program to display the current date and time.

*Sample Output :*

Current date and time :

2014-07-05 14:34:14

Click me to see the sample solution

4. Write a Python program which accepts the radius of a circle from the user and compute the area. Go to the editor

*Sample Output :*

r = 1.1

Area = 3.8013271108436504

[Click me to see the sample solution](#)

**5.** Write a Python program which accepts the user's first and last name and print them in reverse order with a space between them. Go to the editor

[Click me to see the sample solution](#)

**6.** Write a Python program which accepts a sequence of comma-separated numbers from user and generate a list and a tuple with those numbers. Go to the editor

*Sample data : 3, 5, 7, 23*

*Output :*

List : [3, 5, 7, 23]

Tuple : (3, 5, 7, 23)

[Click me to see the sample solution](#)

**7.** Write a Python program to accept a filename from the user and print the extension of that.

Go to the editor

*Sample filename : abc.java*

*Output : java*

[Click me to see the sample solution](#)

**8.** Write a Python program to display the first and last colors from the following list. Go to the editor

color\_list = ["Red","Green","White", "Black"]

[Click me to see the sample solution](#)

**9.** Write a Python program to display the examination schedule. (extract the date from exam\_st\_date). Go to the editor

exam\_st\_date = (11, 12, 2014)

Sample Output : The examination will start from : 11 / 12 / 2014

[Click me to see the sample solution](#)

**10.** Write a Python program that accepts an integer (n) and computes the value of n+nn+nnn.

Go to the editor

*Sample value of n is 5*

*Expected Result : 615*

[Click me to see the sample solution](#)

**11.** Write a Python program to print the documents (syntax, description etc.) of Python built-in function(s).

*Sample function : abs()*

*Expected Result :*

`abs(number) -> number`

Return the absolute value of the argument.

[Click me to see the sample solution](#)

**12.** Write a Python program to print the calendar of a given month and year.

*Note : Use 'calendar' module.*

[Click me to see the sample solution](#)

**13.** Write a Python program to print the following 'here document'. Go to the editor

*Sample string :*

a string that you "don't" have to escape

This

is a ..... multi-line

heredoc string -----> example

[Click me to see the sample solution](#)

**14.** Write a Python program to calculate number of days between two dates.

*Sample dates : (2014, 7, 2), (2014, 7, 11)*

*Expected output : 9 days*

[Click me to see the sample solution](#)

**15.** Write a Python program to get the volume of a sphere with radius 6.

[Click me to see the sample solution](#)

**16.** Write a Python program to get the difference between a given number and 17, if the number is greater than 17 return double the absolute difference. Go to the editor

[Click me to see the sample solution](#)

**17.** Write a Python program to test whether a number is within 100 of 1000 or 2000. Go to the editor

[Click me to see the sample solution](#)

**18.** Write a Python program to calculate the sum of three given numbers, if the values are equal then return three times of their sum. Go to the editor

[Click me to see the sample solution](#)

**19.** Write a Python program to get a new string from a given string where "Is" has been added to the front. If the given string already begins with "Is" then return the string unchanged. Go to the editor

[Click me to see the sample solution](#)

**20.** Write a Python program to get a string which is n (non-negative integer) copies of a given string. Go to the editor

[Click me to see the sample solution](#)

**21.** Write a Python program to find whether a given number (accept from the user) is even or odd, print out an appropriate message to the user. Go to the editor

[Click me to see the sample solution](#)

**22.** Write a Python program to count the number 4 in a given list. Go to the editor

[Click me to see the sample solution](#)

**23.** Write a Python program to get the n (non-negative integer) copies of the first 2 characters of a given string. Return the n copies of the whole string if the length is less than 2. Go to the editor

[Click me to see the sample solution](#)

**24.** Write a Python program to test whether a passed letter is a vowel or not. Go to the editor

[Click me to see the sample solution](#)

**25.** Write a Python program to check whether a specified value is contained in a group of values. Go to the editor

*Test Data :*

3 -> [1, 5, 8, 3] : True

-1 -> [1, 5, 8, 3] : False

[Click me to see the sample solution](#)

**26.** Write a Python program to create a histogram from a given list of integers. Go to the editor

[Click me to see the sample solution](#)

**27.** Write a Python program to concatenate all elements in a list into a string and return it. Go to the editor

[Click me to see the sample solution](#)

- 28.** Write a Python program to print all even numbers from a given numbers list in the same order and stop the printing if any numbers that come after 237 in the sequence. Go to the editor  
*Sample numbers list:*

```
1 numbers = [
2 386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953,
3 399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687,
4 815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742,
5 958, 743, 527
6]
```

[Click me to see the sample solution](#)

- 29.** Write a Python program to print out a set containing all the colors from colorlist\_1 which are not present in color\_list\_2. Go to the editor

*\_Test Data :*

```
colorlist_1 = set(["White", "Black", "Red"])
color_list_2 = set(["Red", "Green"])
```

*\_Expected Output :*

```
{'Black', 'White'}
```

[Click me to see the sample solution](#)

- 30.** Write a Python program that will accept the base and height of a triangle and compute the area. Go to the editor

[Click me to see the sample solution](#)

- 31.** Write a Python program to compute the greatest common divisor (GCD) of two positive integers. Go to the editor

[Click me to see the sample solution](#)

- 32.** Write a Python program to get the least common multiple (LCM) of two positive integers.

[Go to the editor](#)

[Click me to see the sample solution](#)

- 33.** Write a Python program to sum of three given integers. However, if two values are equal sum will be zero. Go to the editor

[Click me to see the sample solution](#)

**34.** Write a Python program to sum of two given integers. However, if the sum is between 15 to 20 it will return 20. Go to the editor

[Click me to see the sample solution](#)

**35.** Write a Python program that will return true if the two given integer values are equal or their sum or difference is 5. Go to the editor

[Click me to see the sample solution](#)

**36.** Write a Python program to add two objects if both objects are an integer type. Go to the editor

[Click me to see the sample solution](#)

**37.** Write a Python program to display your details like name, age, address in three different lines. Go to the editor

[Click me to see the sample solution](#)

**38.** Write a Python program to solve  $(x + y) * (x + y)$ . Go to the editor

*Test Data :* x = 4, y = 3

*Expected Output :*  $(4 + 3)^2 = 49$

[Click me to see the sample solution](#)

**39.** Write a Python program to compute the future value of a specified principal amount, rate of interest, and a number of years. Go to the editor

*Test Data :* amt = 10000, int = 3.5, years = 7

*Expected Output :* 12722.79

[Click me to see the sample solution](#)

**40.** Write a Python program to compute the distance between the points (x1, y1) and (x2, y2).

Go to the editor

[Click me to see the sample solution](#)

**41.** Write a Python program to check whether a file exists. Go to the editor

[Click me to see the sample solution](#)

**42.** Write a Python program to determine if a Python shell is executing in 32bit or 64bit mode on OS. Go to the editor

[Click me to see the sample solution](#)

**43.** Write a Python program to get OS name, platform and release information. Go to the editor  
[Click me to see the sample solution](#)

**44.** Write a Python program to locate Python site-packages. Go to the editor  
[Click me to see the sample solution](#)

**45.** Write a python program to call an external command in Python. Go to the editor  
[Click me to see the sample solution](#)

**46.** Write a python program to get the path and name of the file that is currently executing. Go to the editor  
[Click me to see the sample solution](#)

**47.** Write a Python program to find out the number of CPUs using. Go to the editor  
[Click me to see the sample solution](#)

**48.** Write a Python program to parse a string to Float or Integer. Go to the editor  
[Click me to see the sample solution](#)

**49.** Write a Python program to list all files in a directory in Python. Go to the editor  
[Click me to see the sample solution](#)

**50.** Write a Python program to print without newline or space. Go to the editor  
[Click me to see the sample solution](#)

**51.** Write a Python program to determine profiling of Python programs. Go to the editor  
Note: A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.  
[Click me to see the sample solution](#)

**52.** Write a Python program to print to stderr. Go to the editor  
[Click me to see the sample solution](#)

**53.** Write a python program to access environment variables. Go to the editor  
[Click me to see the sample solution](#)

**54.** Write a Python program to get the current username Go to the editor  
[Click me to see the sample solution](#)

**55.** Write a Python to find local IP addresses using Python's stdlib Go to the editor

[Click me to see the sample solution](#)

**56.** Write a Python program to get height and width of the console window. Go to the editor

[Click me to see the sample solution](#)

**57.** Write a Python program to get execution time for a Python method. Go to the editor

[Click me to see the sample solution](#)

**58.** Write a Python program to sum of the first n positive integers. Go to the editor

[Click me to see the sample solution](#)

**59.** Write a Python program to convert height (in feet and inches) to centimeters. Go to the editor

[Click me to see the sample solution](#)

**60.** Write a Python program to calculate the hypotenuse of a right angled triangle. Go to the editor

[Click me to see the sample solution](#)

**61.** Write a Python program to convert the distance (in feet) to inches, yards, and miles. Go to the editor

[Click me to see the sample solution](#)

**62.** Write a Python program to convert all units of time into seconds. Go to the editor

[Click me to see the sample solution](#)

**63.** Write a Python program to get an absolute file path. Go to the editor

[Click me to see the sample solution](#)

**64.** Write a Python program to get file creation and modification date/times. Go to the editor

[Click me to see the sample solution](#)

**65.** Write a Python program to convert seconds to day, hour, minutes and seconds. Go to the editor

[Click me to see the sample solution](#)

**66.** Write a Python program to calculate body mass index. Go to the editor

[Click me to see the sample solution](#)

**67.** Write a Python program to convert pressure in kilopascals to pounds per square inch, a millimeter of mercury (mmHg) and atmosphere pressure. Go to the editor

[Click me to see the sample solution](#)

**68.** Write a Python program to calculate the sum of the digits in an integer. Go to the editor

[Click me to see the sample solution](#)

**69.** Write a Python program to sort three integers without using conditional statements and loops. Go to the editor

[Click me to see the sample solution](#)

**70.** Write a Python program to sort files by date. Go to the editor

[Click me to see the sample solution](#)

**71.** Write a Python program to get a directory listing, sorted by creation date. Go to the editor

[Click me to see the sample solution](#)

**72.** Write a Python program to get the details of math module. Go to the editor

[Click me to see the sample solution](#)

**73.** Write a Python program to calculate midpoints of a line. Go to the editor

[Click me to see the sample solution](#)

**74.** Write a Python program to hash a word. Go to the editor

[Click me to see the sample solution](#)

**75.** Write a Python program to get the copyright information and write Copyright information in Python code. Go to the editor

[Click me to see the sample solution](#)

**76.** Write a Python program to get the command-line arguments (name of the script, the number of arguments, arguments) passed to a script. Go to the editor

[Click me to see the sample solution](#)

**77.** Write a Python program to test whether the system is a big-endian platform or little-endian platform. Go to the editor

[Click me to see the sample solution](#)

**78.** Write a Python program to find the available built-in modules. Go to the editor

[Click me to see the sample solution](#)

**79.** Write a Python program to get the size of an object in bytes. Go to the editor

[Click me to see the sample solution](#)

**80.** Write a Python program to get the current value of the recursion limit. Go to the editor

[Click me to see the sample solution](#)

**81.** Write a Python program to concatenate N strings. Go to the editor

[Click me to see the sample solution](#)

**82.** Write a Python program to calculate the sum of all items of a container (tuple, list, set, dictionary). Go to the editor

[Click me to see the sample solution](#)

**83.** Write a Python program to test whether all numbers of a list is greater than a certain number. Go to the editor

[Click me to see the sample solution](#)

**84.** Write a Python program to count the number occurrence of a specific character in a string. Go to the editor

[Click me to see the sample solution](#)

**85.** Write a Python program to check whether a file path is a file or a directory. Go to the editor

[Click me to see the sample solution](#)

**86.** Write a Python program to get the ASCII value of a character. Go to the editor

[Click me to see the sample solution](#)

**87.** Write a Python program to get the size of a file. Go to the editor

[Click me to see the sample solution](#)

**88.** Given variables  $x=30$  and  $y=20$ , write a Python program to print "30+20=50". Go to the editor

[Click me to see the sample solution](#)

**89.** Write a Python program to perform an action if a condition is true. Go to the editor

Given a variable name, if the value is 1, display the string "First day of a Month!" and do nothing

if the value is not equal.

[Click me to see the sample solution](#)

**90.** Write a Python program to create a copy of its own source code. Go to the editor

[Click me to see the sample solution](#)

**91.** Write a Python program to swap two variables. Go to the editor

[Click me to see the sample solution](#)

**92.** Write a Python program to define a string containing special characters in various forms.

[Go to the editor](#)

[Click me to see the sample solution](#)

**93.** Write a Python program to get the Identity, Type, and Value of an object. Go to the editor

[Click me to see the sample solution](#)

**94.** Write a Python program to convert a byte string to a list of integers. Go to the editor

[Click me to see the sample solution](#)

**95.** Write a Python program to check whether a string is numeric. Go to the editor

[Click me to see the sample solution](#)

**96.** Write a Python program to print the current call stack. Go to the editor

[Click me to see the sample solution](#)

**97.** Write a Python program to list the special variables used within the language. Go to the editor

[Click me to see the sample solution](#)

**98.** Write a Python program to get the system time. Go to the editor

Note : The system time is important for debugging, network information, random number seeds, or something as simple as program performance.

[Click me to see the sample solution](#)

**99.** Write a Python program to clear the screen or terminal. Go to the editor

[Click me to see the sample solution](#)

**100.** Write a Python program to get the name of the host on which the routine is running. Go to the editor

[Click me to see the sample solution](#)

**101.** Write a Python program to access and print a URL's content to the console. Go to the editor

[Click me to see the sample solution](#)

**102.** Write a Python program to get system command output. Go to the editor

[Click me to see the sample solution](#)

**103.** Write a Python program to extract the filename from a given path. Go to the editor

[Click me to see the sample solution](#)

**104.** Write a Python program to get the effective group id, effective user id, real group id, a list of supplemental group ids associated with the current process. Go to the editor

Note: Availability: Unix.

[Click me to see the sample solution](#)

**105.** Write a Python program to get the users environment. Go to the editor

[Click me to see the sample solution](#)

**106.** Write a Python program to divide a path on the extension separator. Go to the editor

[Click me to see the sample solution](#)

**107.** Write a Python program to retrieve file properties. Go to the editor

[Click me to see the sample solution](#)

**108.** Write a Python program to find path refers to a file or directory when you encounter a path name. Go to the editor

[Click me to see the sample solution](#)

**109.** Write a Python program to check if a number is positive, negative or zero. Go to the editor

[Click me to see the sample solution](#)

**110.** Write a Python program to get numbers divisible by fifteen from a list using an anonymous function. Go to the editor

[Click me to see the sample solution](#)

**111.** Write a Python program to make file lists from current directory using a wildcard. Go to the editor

[Click me to see the sample solution](#)

**112.** Write a Python program to remove the first item from a specified list. Go to the editor

[Click me to see the sample solution](#)

**113.** Write a Python program to input a number, if it is not a number generates an error message. Go to the editor

[Click me to see the sample solution](#)

**114.** Write a Python program to filter the positive numbers from a list. Go to the editor

[Click me to see the sample solution](#)

**115.** Write a Python program to compute the product of a list of integers (without using for loop). Go to the editor

[Click me to see the sample solution](#)

**116.** Write a Python program to print Unicode characters. Go to the editor

[Click me to see the sample solution](#)

**117.** Write a Python program to prove that two string variables of same value point same memory location. Go to the editor

[Click me to see the sample solution](#)

**118.** Write a Python program to create a bytearray from a list. Go to the editor

[Click me to see the sample solution](#)

**119.** Write a Python program to round a floating-point number to specified number decimal places. Go to the editor

[Click me to see the sample solution](#)

**120.** Write a Python program to format a specified string limiting the length of a string. Go to the editor

[Click me to see the sample solution](#)

**121.** Write a Python program to determine whether variable is defined or not. Go to the editor

[Click me to see the sample solution](#)

**122.** Write a Python program to empty a variable without destroying it. Go to the editor

Sample data: n=20

```
d = {"x":200}
```

Expected Output : 0

```
{}
```

[Click me to see the sample solution](#)

**123.** Write a Python program to determine the largest and smallest integers, longs, floats. Go to the editor

[Click me to see the sample solution](#)

**124.** Write a Python program to check whether multiple variables have the same value. Go to the editor

[Click me to see the sample solution](#)

**125.** Write a Python program to sum of all counts in a collections. Go to the editor

[Click me to see the sample solution](#)

**126.** Write a Python program to get the actual module object for a given object. Go to the editor

[Click me to see the sample solution](#)

**127.** Write a Python program to check whether an integer fits in 64 bits. Go to the editor

[Click me to see the sample solution](#)

**128.** Write a Python program to check whether lowercase letters exist in a string. Go to the editor

[Click me to see the sample solution](#)

**129.** Write a Python program to add leading zeroes to a string. Go to the editor

[Click me to see the sample solution](#)

**130.** Write a Python program to use double quotes to display strings. Go to the editor

[Click me to see the sample solution](#)

**131.** Write a Python program to split a variable length string into variables. Go to the editor

[Click me to see the sample solution](#)

**132.** Write a Python program to list home directory without absolute path. Go to the editor  
[Click me to see the sample solution](#)

**133.** Write a Python program to calculate the time runs (difference between start and current time) of a program. Go to the editor  
[Click me to see the sample solution](#)

**134.** Write a Python program to input two integers in a single line. Go to the editor  
[Click me to see the sample solution](#)

**135.** Write a Python program to print a variable without spaces between values. Go to the editor  
Sample value : x =30

Expected output : Value of x is "30"  
[Click me to see the sample solution](#)

**136.** Write a Python program to find files and skip directories of a given directory. Go to the editor  
[Click me to see the sample solution](#)

**137.** Write a Python program to extract single key-value pair of a dictionary in variables. Go to the editor  
[Click me to see the sample solution](#)

**138.** Write a Python program to convert true to 1 and false to 0. Go to the editor  
[Click me to see the sample solution](#)

**139.** Write a Python program to valid a IP address. Go to the editor  
[Click me to see the sample solution](#)

**140.** Write a Python program to convert an integer to binary keep leading zeros. Go to the editor  
Sample data : x=12  
Expected output : 00001100  
0000001100  
[Click me to see the sample solution](#)

**141.** Write a python program to convert decimal to hexadecimal. Go to the editor  
Sample decimal number: 30, 4  
Expected output: 1e, 04  
[Click me to see the sample solution](#)

**142.** Write a Python program to find the operating system name, platform and platform release date. Go to the editor

Operating system name:

posix

Platform name:

Linux

Platform release:

4.4.0-47-generic

[Click me to see the sample solution](#)

**143.** Write a Python program to determine if the python shell is executing in 32bit or 64bit mode on operating system. Go to the editor

[Click me to see the sample solution](#)

**144.** Write a Python program to check whether variable is integer or string. Go to the editor

[Click me to see the sample solution](#)

**145.** Write a Python program to test if a variable is a list or tuple or a set. Go to the editor

[Click me to see the sample solution](#)

**146.** Write a Python program to find the location of Python module sources. Go to the editor

[Click me to see the sample solution](#)

**147.** Write a Python function to check whether a number is divisible by another number. Accept two integers values form the user. Go to the editor

[Click me to see the sample solution](#)

**148.** Write a Python function to find the maximum and minimum numbers from a sequence of numbers. Go to the editor

Note: Do not use built-in functions.

[Click me to see the sample solution](#)

**149.** Write a Python function that takes a positive integer and returns the sum of the cube of all the positive integers smaller than the specified number. Go to the editor

[Click me to see the sample solution](#)

**150.** Write a Python function to check whether a distinct pair of numbers whose product is odd present in a sequence of integer values. Go to the editor

[Click me to see the sample solution](#)

## Direct

```
1 import math
2
3
4 def say_hi(name):
5 """<---- Multi-Line Comments and Docstrings
6 This is where you put your content for help() to inform the user
7 about what your function does and how to use it
8 """
9 print(f"Hello {name}!")
10
11
12 print(say_hi("Bryan")) # Should get the print inside the function, then
13 # Boolean Values
14 # Work the same as in JS, except they are title case: True and False
15 a = True
16 b = False
17 # Logical Operators
18 # ! = not, || = or, && = and
19 print(True and True)
20 print(True and not True)
21 print(True or True)
22 # Truthiness - Everything is True except...
23 # False - None, False, '', [], (), set(), range(0)
24 # Number Values
25 # Integers are numbers without a floating decimal point
26 print(type(3)) # type returns the type of whatever argument you pass in
27 # Floating Point values are numbers with a floating decimal point
28 print(type(3.5))
29 # Type Casting
30 # You can convert between ints and floats (along with other types...)
31 print(float(3)) # If you convert a float to an int, it will truncate the
32 print(int(4.5))
33 print(type(str(3)))
34 # Python does not automatically convert types like JS
35 # print(17.0 + ' heyooo ' + 17) # TypeError
36 # Arithmetic Operators
37 # ** - exponent (comparable to Math.pow(num, pow))
38 # // - integer division
39 # There is no ++ or -- in Python
40 # String Values
41 # We can use single quotes, double quotes, or f'' for string formats
42 # We can use triple single quotes for multiline strings
43 print(
44 """This here's a story
45 All about how
46 My life got twist
47 Turned upside down
48 """
```

```
49)
50 # Three double quotes can also be used, but we typically reserve these
51 # multi-line comments and function docstrings (refer to lines 6-9)(Nice)
52 # We use len() to get the length of something
53 print(len("Bryan G")) # 7 characters
54 print(len(["hey", "ho", "hey", "hey", "ho"])) # 5 list items
55 print(len({1, 2, 3, 4, 5, 6, 7, 9})) # 8 set items
56 # We can index into strings, list, etc..self.
57 name = "Bryan"
58 for i in range(len(name)):
59 print(name[i]) # B, r, y, a, n
60 # We can index starting from the end as well, with negatives
61 occupation = "Full Stack Software Engineer"
62 print(occupation[-3]) # e
63 # We can also get ranges in the index with the [start:stop:step] syntax
64 print(occupation[0:4:1]) # step and stop are optional, stop is exclusiv
65 print(occupation[::-4]) # beginning to end, every 4th letter
66 print(occupation[4:14:2]) # Let's get weird with it!
67 # NOTE: Indexing out of range will give you an IndexError
68 # We can also get the index og things with the .index() method, similar
69 print(occupation.index("Stack"))
70 print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
71 # We can count how many times a substring/item appears in something as w
72 print(occupation.count("S"))
73 print(
74 """Now this here's a story all about how
75 My life got twist turned upside down
76 I forget the rest but the the potato
77 smells like the potato""".count(
78 "the"
79)
80)
81 # We concatenate the same as Javascript, but we can also multiply strings
82 print("dog " + "show")
83 print("ha" * 10)
84 # We can use format for a multitude of things, from spaces to decimal p
85 first_name = "Bryan"
86 last_name = "Guner"
87 print("Your name is {0} {1}".format(first_name, last_name))
88 # Useful String Methods
89 print("Hello".upper()) # HELLO
90 print("Hello".lower()) # hello
91 print("HELLO".islower()) # False
92 print("HELLO".isupper()) # True
93 print("Hello".startswith("he")) # False
94 print("Hello".endswith("lo")) # True
95 print("Hello There".split()) # [Hello, There]
96 print("hello1".isalpha()) # False, must consist only of letters
97 print("hello1".isalnum()) # True, must consist of only letters and numbe
98 print("3215235123".isdecimal()) # True, must be all numbers
99 # True, must consist of only spaces/tabs/newlines
100 print("\n ".isspace())
101 # False, index 0 must be upper case and the rest lower
```

```
102 print("Bryan Guner".istitle())
103 print("Michael Lee".istitle()) # True!
104 # Duck Typing - If it walks like a duck, and talks like a duck, it must
105 # Assignment - All like JS, but there are no special keywords like let or const
106 a = 3
107 b = a
108 c = "heyoo"
109 b = ["reassignment", "is", "fine", "G!"]
110 # Comparison Operators - Python uses the same equality operators as JS,
111 # < - Less than
112 # > - Greater than
113 # <= - Less than or Equal
114 # >= - Greater than or Equal
115 # == - Equal to
116 # != - Not equal to
117 # is - Refers to exact same memory location
118 # not - !
119 # Precedence - Negative Signs(not) are applied first(part of each number)
120 # - Multiplication and Division(and) happen next
121 # - Addition and Subtraction(or) are the last step
122 # NOTE: Be careful when using not along with ==
123 print(not a == b) # True
124 # print(a == not b) # Syntax Error
125 print(a == (not b)) # This fixes it. Answer: False
126 # Python does short-circuit evaluation
127 # Assignment Operators - Mostly the same as JS except Python has **=
128 # Flow Control Statements - if, while, for
129 # Note: Python smushes 'else if' into 'elif'!
130 if 10 < 1:
131 print("We don't get here")
132 elif 10 < 5:
133 print("Nor here...")
134 else:
135 print("Hey there!")
136 # Looping over a string
137 for c in "abcdefghijklmnopqrstuvwxyz":
138 print(c)
139 # Looping over a range
140 for i in range(5):
141 print(i + 1)
142 # Looping over a list
143 lst = [1, 2, 3, 4]
144 for i in lst:
145 print(i)
146 # Looping over a dictionary
147 spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
148 for v in spam.values():
149 print(v)
150 # Loop over a list of tuples and destructuring the values
151 # Assuming spam.items returns a list of tuples each containing two items
152 for k, v in spam.items():
153 print(f"{k}: {v}")
154 # While loops as long as the condition is True
```

```
155 # - Exit loop early with break
156 # - Exit iteration early with continue
157 spam = 0
158 while True:
159 print("Sike That's the wrong Numba")
160 spam += 1
161 if spam < 5:
162 continue
163 break
164
165 # Functions - use def keyword to define a function in Python
166
167
168 def printCopyright():
169 print("Copyright 2021, Bgoonz")
170
171
172 # Lambdas are one liners! (Should be at least, you can use parenthesis)
173 def avg(num1, num2):
174 return print(num1 + num2)
175
176
177 avg(1, 2)
178 # Calling it with keyword arguments, order does not matter
179 avg(num2=20, num1=1252)
180 printCopyright()
181 # We can give parameters default arguments like JS
182
183
184 def greeting(name, saying="Hello"):
185 print(saying, name)
186
187
188 greeting("Mike") # Hello Mike
189 greeting("Bryan", saying="Hello there...")
190 # A common gotcha is using a mutable object for a default parameter
191 # All invocations of the function reference the same mutable object
192
193
194 def append_item(item_name, item_list=[]): # Will it obey and give us a
195 item_list.append(item_name)
196 return item_list
197
198
199 # Uses same item list unless otherwise stated which is counterintuitive
200 print	append_item("notebook"))
201 print	append_item("notebook"))
202 print	append_item("notebook", []))
203 # Errors - Unlike JS, if we pass the incorrect amount of arguments to a
204 # it will throw an error
205 # avg(1) # TypeError
206 # avg(1, 2, 2) # TypeError
207 # ----- DAY 2 -----
```

```

208 # Functions - * to get rest of position arguments as tuple
209 # - ** to get rest of keyword arguments as a dictionary
210 # Variable Length positional arguments
211
212
213 def add(a, b, *args):
214 # args is a tuple of the rest of the arguments
215 total = a + b
216 for n in args:
217 total += n
218 return total
219
220
221 print(add(1, 2)) # args is None, returns 3
222 print(add(1, 2, 3, 4, 5, 6)) # args is (3, 4, 5, 6), returns 21
223 # Variable Length Keyword Arguments
224
225
226 def print_names_and_countries(greeting, **kwargs):
227 # kwargs is a dictionary of the rest of the keyword arguments
228 for k, v in kwargs.items():
229 print(greeting, k, "from", v)
230
231
232 print_names_and_countries(
233 "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
234)
235 # We can combine all of these together
236
237
238 def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
239 pass
240
241
242 # Lists are mutable arrays
243 empty_list = []
244 roomates = ["Beau", "Delynn"]
245 # List built-in function makes a list too
246 specials = list()
247 # We can use 'in' to test if something is in the list, like 'includes'
248 print(1 in [1, 2, 4]) # True
249 print(2 in [1, 3, 5]) # False
250 # Dictionaries - Similar to JS POJO's or Map, containing key value pairs
251 a = {"one": 1, "two": 2, "three": 3}
252 b = dict(one=1, two=2, three=3)
253 # Can use 'in' on dictionaries too (for keys)
254 print("one" in a) # True
255 print(3 in b) # False
256 # Sets - Just like JS, unordered collection of distinct objects
257 bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}
258 # bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
259 school_bag = set(
260 ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"]

```

```
261)
262 print(school_bag)
263 print(bedroom)
264 # We can use 'in' on sets as well
265 print(1 in {1, 2, 3}) # True
266 print(4 in {1, 3, 5}) # False
267 # Tuples are immutable lists of items
268 time_blocks = ("AM", "PM")
269 colors = "red", "green", "blue" # Parenthesis not needed but encouraged
270 # The tuple built-in function can be used to convert things to tuples
271 print(tuple("abc"))
272 print(tuple([1, 2, 3]))
273 # 'in' may be used on tuples as well
274 print(1 in (1, 2, 3)) # True
275 print(5 in (1, 4, 3)) # False
276 # Ranges are immutable lists of numbers, often used with for loops
277 # - start - default: 0, first number in sequence
278 # - stop - required, next number past last number in sequence
279 # - step - default: 1, difference between each number in sequence
280 range1 = range(5) # [0,1,2,3,4]
281 range2 = range(1, 5) # [1,2,3,4]
282 range3 = range(0, 25, 5) # [0,5,10,15,20]
283 range4 = range(0) # []
284 for i in range1:
285 print(i)
286 # Built-in functions:
287 # Filter
288
289
290 def isOdd(num):
291 return num % 2 == 1
292
293
294 filtered = filter(isOdd, [1, 2, 3, 4])
295 print(list(filtered))
296 for num in filtered:
297 print(f"first way: {num}")
298 print("--" * 20)
299 [print(f"list comprehension: {i}")
300 for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
301 # Map
302
303
304 def toUpper(str):
305 return str.upper()
306
307
308 upperCased = map(toUpper, ["a", "b", "c", "d"])
309 print(list(upperCased))
310 # Sorted
311 sorted_items = sorted(["john", "tom", "sonny", "Mike"])
312 print(list(sorted_items)) # Notice uppercase comes before lowercase
313 # Using a key function to control the sorting and make it case insensitive
```

```
314 sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
315 print(sorted_items)
316 # You can also reverse the sort
317 sorted_items = sorted(["john", "tom", "sonny", "Mike"],
318 key=str.lower, reverse=True)
319 print(sorted_items)
320 # Enumerate creates a tuple with an index for what you're enumerating
321 quarters = ["First", "Second", "Third", "Fourth"]
322 print(list(enumerate(quarters)))
323 print(list(enumerate(quarters, start=1)))
324 # Zip takes list and combines them as key value pairs, or really however
325 keys = ("Name", "Email")
326 values = ("Buster", "cheetoh@johhnydepp.com")
327 zipped = zip(keys, values)
328 print(list(zipped))
329 # You can zip more than 2
330 x_coords = [0, 1, 2, 3, 4]
331 y_coords = [4, 6, 10, 9, 10]
332 z_coords = [20, 10, 5, 9, 1]
333 coords = zip(x_coords, y_coords, z_coords)
334 print(list(coords))
335 # Len reports the length of strings along with list and any other object
336 # doing this to save myself some typing
337
338
339 def print_len(item):
340 return print(len(item))
341
342
343 print_len("Mike")
344 print_len([1, 5, 2, 10, 3, 10])
345 print_len({1, 5, 10, 9, 10}) # 4 because there is a duplicate here (10)
346 print_len((1, 4, 10, 9, 20))
347 # Max will return the max number in a given scenario
348 print(max(1, 2, 35, 1012, 1))
349 # Min
350 print(min(1, 5, 2, 10))
351 print(min([1, 4, 7, 10]))
352 # Sum
353 print(sum([1, 2, 4]))
354 # Any
355 print(any([True, False, False]))
356 print(any([False, False, False]))
357 # All
358 print(all([True, True, False]))
359 print(all([True, True, True]))
360 # Dir returns all the attributes of an object including it's methods and
361 user = {"Name": "Bob", "Email": "bob@bob.com"}
362 print(dir(user))
363 # Importing packages and modules
364 # - Module - A Python code in a file or directory
365 # - Package - A module which is a directory containing an __init__.py
366 # - Submodule - A module which is contained within a package
```

```
367 # - Name - An exported function, class, or variable in a module
368 # Unlike JS, modules export ALL names contained within them without any
369 # Assuming we have the following package with four submodules
370 # math
371 # | __init__.py
372 # | addition.py
373 # | subtraction.py
374 # | multiplication.py
375 # | division.py
376 # If we peek into the addition.py file we see there's an add function
377 # addition.py
378 # We can import 'add' from other places because it's a 'name' and is au-
379
380
381 # def add(num1, num2):
382 # return num1 + num2
383
384
385 # Notice the . syntax because this package can import it's own submodule
386 # Our __init__.py has the following files
387 # This imports the 'add' function
388 # And now it's also re-exported in here as well
389 # from .addition import add
390 # These import and re-export the rest of the functions from the submodule
391 # from .subtraction import subtract
392 # from .division import divide
393 # from .multiplication import multiply
394 # So if we have a script.py and want to import add, we could do it many
395 # This will load and execute the 'math/__init__.py' file and give
396 # us an object with the exported names in 'math/__init__.py'
397 # print(math.add(1,2))
398 # This imports JUST the add from 'math/__init__.py'
399 # from math import add
400 # print(add(1, 2))
401 # This skips importing from 'math/__init__.py' (although it still runs)
402 # and imports directly from the addition.py file
403 # from math.addition import add
404 # This imports all the functions individually from 'math/__init__.py'
405 # from math import add, subtract, multiply, divide
406 # print(add(1, 2))
407 # print(subtract(2, 1))
408 # This imports 'add' renames it to 'add_some_numbers'
409 # from math import add as add_some_numbers
410 # ----- DAY 3 -----
411 # Classes, Methods, and Properties
412
413
414 class AngryBird:
415 # Slots optimize property access and memory usage and prevent you
416 # from arbitrarily assigning new properties to the instance
417 __slots__ = ["_x", "_y"]
418 # Constructor
419
```

```
420 def __init__(self, x=0, y=0):
421 # Doc String
422 """
423 Construct a new AngryBird by setting it's position to (0, 0)
424 """
425 # Instance Variables
426 self._x = x
427 self._y = y
428
429 # Instance Method
430
431 def move_up_by(self, delta):
432 self._y += delta
433
434 # Getter
435
436 @property
437 def x(self):
438 return self._x
439
440 # Setter
441
442 @x.setter
443 def x(self, value):
444 if value < 0:
445 value = 0
446 self._x = value
447
448 @property
449 def y(self):
450 return self._y
451
452 @y.setter
453 def y(self, value):
454 self._y = value
455
456 # Dunder Repr... called by 'print'
457
458 def __repr__(self):
459 return f"<AngryBird ({self._x}, {self._y})>"
460
461
462 # JS to Python Classes cheat table
463 # JS Python
464 # constructor() def __init__(self):
465 # super() super().__init__()
466 # this.property self.property
467 # this.method self.method()
468 # method(arg1, arg2){} def method(self, arg1, ...)
469 # get someProperty(){} @property
470 # set someProperty(){} @someProperty.setter
471 # List Comprehensions are a way to transform a list from one format to another
472 # - Pythonic Alternative to using map or filter
```

```
473 # - Syntax of a list comprehension
474 # - new_list = [value loop condition]
475 # Using a for loop
476 squares = []
477 for i in range(10):
478 squares.append(i ** 2)
479 print(squares)
480 # value = i ** 2
481 # loop = for i in range(10)
482 squares = [i ** 2 for i in range(10)]
483 print(list(squares))
484 sentence = "the rocket came back from mars"
485 vowels = [character for character in sentence if character in "aeiou"]
486 print(vowels)
487 # You can also use them on dictionaries. We can use the items() method
488 # for the dictionary to loop through it getting the keys and values out
489 person = {"name": "Corina", "age": 32, "height": 1.4}
490 # This loops through and capitalizes the first letter of all keys
491 newPerson = {key.title(): value for key, value in person.items()}
492 print(list(newPerson.items()))
```

## Gist

## Repl



StripedInternationalMotion

<https://replit.com/@bgoonz/StripedInternationalMotion#main.py>

# Untitled

# Generate a graph

Difficulty Level : Medium

- Last Updated : 28 Jun, 2021

Prerequisite – Graphs

To draw graph using in built libraries – Graph plotting in Python

In this article, we will see how to implement graph in python using dictionary data structure in python.

The keys of the dictionary used are the nodes of our graph and the corresponding values are lists with each nodes, which are connecting by an edge.

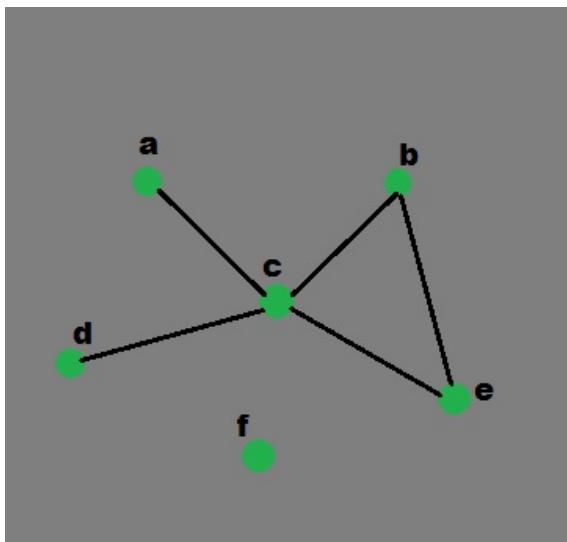
This simple graph has six nodes (a-f) and five arcs:

```
1 a -> c
2 b -> c
3 b -> e
4 c -> a
5 c -> b
6 c -> d
7 c -> e
8 d -> c
9 e -> c
10 e -> b
```

It can be represented by the following Python data structure. This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node.

```
1 graph = { "a" : ["c"],
2 "b" : ["c", "e"],
3 "c" : ["a", "b", "d", "e"],
4 "d" : ["c"],
5 "e" : ["c", "b"],
6 "f" : []}
```

**Graphical representation of above example:**



`defaultdict`: Usually, a Python dictionary throws a `KeyError` if you try to get an item with a key that is not currently in the dictionary. `defaultdict` allows that if a key is not found in the dictionary, then instead of a `KeyError` being thrown, a new entry is created. The type of this new entry is given by the argument of `defaultdict`.

#### Python Function to generate graph:

```

1 # definition of function
2 def generate_edges(graph):
3 edges = []
4
5 # for each node in graph
6 for node in graph:
7
8 # for each neighbour node of a single node
9 for neighbour in graph[node]:
10 # if edge exists then append
11 edges.append((node, neighbour))
12
13 return edges

```

Recommended: Please try your approach on **{IDE}** first, before moving on to the solution.

## Python

Output:

```
1 [('a', 'c'), ('c', 'd'), ('c', 'e'), ('c', 'a'), ('c', 'b'),
2 ('b', 'c'), ('b', 'e'), ('e', 'b'), ('e', 'c'), ('d', 'c')]
```

As we have taken example of undirected graph, so we have print same edge twice say as ('a','c') and ('c','a'). We can overcome this with use of directed graph.

Below are some more programs on graphs in python:

1. **To generate the path from one node to the other node:** Using Python dictionary, we can find the path from one node to the other in a Graph. The idea is similar to DFS in graphs. In the function, initially, the path is an empty list. In the starting, if the start node matches with the end node, the function will return the path. Otherwise the code goes forward and hits all the values of the starting node and searches for the path using recursion.

## Python



blank-1

<https://replit.com/@bgoonz/blank-1#main.py>

1. Output:

```
['d', 'a', 'c']
```

1. **Program to generate all the possible paths from one node to the other.:** In the above discussed program, we generated the first possible path. Now, let us generate all the possible paths from the start node to the end node. The basic functioning works same as the functioning of the above code. The place where the difference comes is instead of instantly returning the first path, it saves that path in a list named as 'paths' in the example given below. Finally, after iterating over all the possible ways, it returns the list of paths. If there is no path from the starting node to the ending node, it returns None.

# Python

possible-paths.py

<https://gist.github.com/bgoonz/b46192bc87d3cac28bdcf392afbe842d>

## Output:

```
[['d', 'a', 'c'], ['d', 'a', 'c']]
```

1. **Program to generate the shortest path.**: To get to the shortest from all the paths, we use a little different approach as shown below. In this, as we get the path from the start node to the end node, we compare the length of the path with a variable named as shortest which is initialized with the None value. If the length of generated path is less than the length of shortest, if shortest is not None, the newly generated path is set as the value of shortest. Again, if there is no path, it returns None

## 2. Python

```
Python program to generate shortest path
graph = {'a': ['c'], 'b': ['d'], 'c': ['e'], 'd': ['a', 'd'], 'e': ['b', 'c']}
function to find the shortest path
def find_shortest_path(graph, start, end, path = []):
 path = path + [start]
 if start == end:
 return path
 shortest = None
 for node in graph[start]:
 if node not in path:
 newpath = find_shortest_path(graph, node, end, path)
 if newpath:
 if not shortest or len(newpath) < len(shortest):
 shortest = newpath
 return shortest
Driver function call to print# the shortest
pathprint(find_shortest_path(graph,
'd', 'c'))
```

## 1. Output:

['d', 'a', 'c']

# All Along

# **Beginners Guide To Python**

My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library support, and optional...

---

## **Beginners Guide To Python**

**My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library support, and optional named parameters.**

| Bram Cohen

**Article on basic web development setup... it is geared towards web but VSCode is an incredibly versatile editor and this stack really could suit just about anyone working in the field of computer science.**



sys Variables		String Methods		Datetime Methods	
argv	Command line args	capitalize() *	lstrip()	today()	fromordinal(ordinal)
builtin_module_names	Linked C modules	center(width)	partition(sep)	now(timezoneinfo)	combine(date, time)
byteorder	Native byte order	count(sub, start, end)	replace(old, new)	utcnow()	strftime(date, format)
check_interval	Signal check frequency	decode()	rfind(sub, start, end)	fromtimestamp(timestamp)	utcfromtimestamp(timestamp)
exec_prefix	Root directory	encode()	rindex(sub, start, end)		
executable	Name of executable	endswith(sub)	rjust(width)		
exitfunc	Exit function name	expandtabs()	rpartition(sep)		
modules	Loaded modules	find(sub, start, end)	rsplit(sep)		
path	Search path	index(sub, start, end)	rstrip()		
platform	Current platform	isalnum() *	split(sep)		
stdin, stdout, stderr	File objects for I/O	isalpha() *	splines()		
version_info	Python version info	isdigit() *	startswith(sub)		
winver	Version number	islower() *	strip()		
<b>sys.argv for \$ python foo.py bar -c qux --h</b>		isspace() *	swapcase() *		
sys.argv[0]	foo.py	istitle() *	title() *		
sys.argv[1]	bar	isupper() *	translate(table)		
sys.argv[2]	-c	join()	upper() *		
sys.argv[3]	qux	ljust(width)	zfill(width)		
sys.argv[4]	--h	lower() *			
<b>Note</b>		Methods marked * are locale dependant for 8-bit strings.			
os Variables		List Methods		Time Methods	
altsep	Alternative sep	append(item)	pop(position)	replace()	utcoffset()
curdir	Current dir string	count(item)	remove(item)	isoformat()	dst()
defpath	Default search path	extend(list)	reverse()	__str__()	tzname()
devnull	Path of null device	index(item)	sort()	strftime(format)	
extsep	Extension separator	insert(position, item)			
linesep	Line separator				
name	Name of OS				
pardir	Parent dir string				
pathsep	Patch separator				
sep	Path separator				
<b>Note</b>					
Registered OS names: "posix", "nt", "mac", "os2", "ce", "java", "riscos"					
Class Special Methods		File Methods		Date Formatting (strftime and strptime)	
__new__(cls)	__lt__(self, other)	close()	readlines(size)	%a Abbreviated weekday (Sun)	
__init__(self, args)	__le__(self, other)	flush()	seek(offset)	%A Weekday (Sunday)	
__del__(self)	__gt__(self, other)	fileno()	tell()	%b Abbreviated month name (Jan)	
__repr__(self)	__ge__(self, other)	isatty()	truncate(size)	%B Month name (January)	
__str__(self)	__eq__(self, other)	next()	write(string)	%c Date and time	
__cmp__(self, other)	__ne__(self, other)	read(size)	writelines(list)	%d Day (leading zeros) (01 to 31)	
__index__(self)	__nonzero__(self)	readline(size)		%H 24 hour (leading zeros) (00 to 23)	
__hash__(self)				%I 12 hour (leading zeros) (01 to 12)	
__getattr__(self, name)				%j Day of year (001 to 366)	
__getattribute__(self, name)				%m Month (01 to 12)	
__setattr__(self, name, attr)				%M Minute (00 to 59)	
__delattr__(self, name)				%p AM or PM	
__call__(self, args, kwargs)				%S Second (00 to 61*)	
				%U Week number ^ (00 to 53)	
				%w Weekday ^ (0 to 6)	
				%W Week number ^ (00 to 53)	
				%x Date	
				%X Time	
				%y Year without century (00 to 99)	
				%Y Year (2008)	
				%Z Time zone (GMT)	
				%% A literal "%" character (%)	
<b>Indexes and Slices (of a=[0,1,2,3,4,5])</b>		1. Sunday as start of week. All days in a new year preceding the first Sunday are considered to be in week 0.		2. 0 is Sunday, 6 is Saturday.	
len(a)	6	a[0]	0	3. Monday as start of week. All days in a new year preceding the first Monday are considered to be in week 0.	
a[0]	0	a[5]	5	4. This is not a mistake. Range takes account of leap and double-leap seconds.	
a[-1]	5	a[-2]	4		
a[1:-1]	[1,2,3,4,5]	a[1:3]	[1,2]		
a[:5]	[0,1,2,3,4]	a[1:-1]	[1,2,3,4]		
a[:-2]	[0,1,2,3]	b=a[:]	Shallow copy of a		

Available free from [AddedBytes.com](http://AddedBytes.com)

## Python

- Python is an interpreted, high-level and general-purpose, dynamically typed programming language
- It is also Object oriented, modular oriented and a scripting language.
- In Python, everything is considered as an Object.
- A python file has an extension of .py
- Python follows Indentation to separate code blocks instead of flower brackets({}).
- We can run a python file by the following command in cmd(Windows) or shell(mac/linux).
- `python <filename.py>`

**By default, the python doesn't require any imports to run a python file.**

## Create and execute a program

1. Open up a terminal/cmd
  2. Create the program: nano/cat > nameProgram.py
  3. Write the program and save it
  4. python nameProgram.py
- 

## Basic Datatypes

### Basic Datatypes

Data Type	Description
int	Integer values [0, 1, -2, 3]
float	Floating point values [0.1, 4.532, -5.092]
char	Characters [a, b, @, !, `]
str	Strings [abc, AbC, A@B, sd!, `asa]
bool	Boolean Values [True, False]
char	Characters [a, b, @, !, `]
complex	Complex numbers [2+3j, 4-1j]

Data Type Description  
int Integer values [0, 1, -2, 3]  
float Floating point values [0.1, 4.532, -5.092]  
char Characters [a, b, @, !, `]  
str Strings [abc, AbC, A@B, sd!, `asa]  
bool Boolean Values [True, False]  
char Characters [a, b, @, !, `]  
complex Complex numbers [2+3j, 4-1j]

---

## Keywords

## Keywords

Keyword	Description
break	used to exit loop and used to exit
char	basic declaration of a type character
const	prefix declaration meaning variable can not be changed
continue	go to bottom of loop in for, while loops
class	to define a class
def	to define a function
elif	shortcut for (else if) used in else if ladder
else	executable statement, part of "if" structure
float	basic declaration of floating point
for	executable statement, for loop
from	executable statement, used to import only specific objects from a package
if	executable statement
import	to import modules
pass	keyword to specify nothing is happening in the codeblock, generally used in classes
return	executable statement with or without a value
while	executable statement, while loop

KeywordDescription  
breakused to exit loop and used to exit  
charbasic declaration of a type  
character  
constprefix declaration meaning variable can not be changed  
continuego to bottom of  
loop in for, while loops  
class to define a class  
defto define a function  
elifshortcut for (else if) used in else if ladder  
elseexecutable statement, part of "if" structure  
floatbasic declaration of floating  
point  
forexecutable statement, for loop  
fromexecutable statement, used to import only specific  
objects from a package  
ife

---

## Operators

## Operators

Operator	Description
( )	grouping parenthesis, function call, tuple declaration
[ ]	array indexing, also declaring lists etc.
!	relational not, complement, ! a yields true or false
~	bitwise not, ones complement, ~a
-	unary minus, - a
+	unary plus, + a
*	multiply, a * b
/	divide, a / b
%	modulo, a % b
+	add, a + b
-	subtract, a - b
<<	shift left, left operand is shifted left by right operand bits
>>	shift right, left operand is shifted right by right operand bits
<	less than, result is true or false, a < b
<=	less than or equal, result is true or false, a <= b

>	greater than, result is true or false, a > b
>=	greater than or equal, result is true or false, a >= b
==	equal, result is true or false, a == b
!=	not equal, result is true or false, a != b
&	bitwise and, a & b
^	bitwise exclusive or XOR, a ^ b
	bitwise or, a
&&, and	relational and, result is true or false, a < b && c >= d
, or	relational or, result is true or false, a < b    c >= d
=	store or assignment
+=	add and store
-=	subtract and store
*=	multiply and store
/=	divide and store
%=	modulo and store
<<=	shift left and store
>>=	shift right and store
&=	bitwise and and store
^=	bitwise exclusive or and store
=	bitwise or and store
,	separator as in ( y=x,z=++x )

## Basic Data Structures

### List

- List is a collection which is ordered and changeable. Allows duplicate members.
- Lists are created using square brackets:

```
thislist = ["apple", "banana", "cherry"]
```

- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0] , the second item has index [1] etc.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- To determine how many items a list has, use the len() function.

- A list can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
```

- It is also possible to use the list() constructor when creating a new list

```
thislist = list(("apple", "banana", "cherry")) # note the double round-bracket
```

## Tuple

- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.

```
thistuple = ("apple", "banana", "cherry")
```

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0] , the second item has index [1] etc.
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Since tuple are indexed, tuples can have items with the same value:
- Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
```

- To determine how many items a tuple has, use the len() function:

```
1 thistuple = ("apple", "banana", "cherry")
2 print(len(thistuple))
```

- To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
1 thistuple = ("apple",)
2 print(type(thistuple))
```

```
1 #NOT a tuple
2 thistuple = ("apple")
3 print(type(thistuple))
```

- It is also possible to use the tuple() constructor to make a tuple.

```
1 thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
2 print(thistuple)
```

## Set

- Set is a collection which is unordered and unindexed. No duplicate members.
- A set is a collection which is both unordered and unindexed.

```
thisset = {"apple", "banana", "cherry"}
```

- Set items are unordered, unchangeable, and do not allow duplicate values.
- Unordered means that the items in a set do not have a defined order.

- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Sets are unchangeable, meaning that we cannot change the items after the set has been created.
- Duplicate values will be ignored.
- To determine how many items a set has, use the `len()` method.

```
thisset = {"apple", "banana", "cherry"}
```

```
print(len(thisset))
```

- Set items can be of any data type:

```
1 set1 = {"apple", "banana", "cherry"}
2 set2 = {1, 5, 7, 9, 3}
3 set3 = {True, False, False}
4 set4 = {"abc", 34, True, 40, "male"}
```

- It is also possible to use the `set()` constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
```

## Dictionary

- Dictionary is a collection which is unordered and changeable. No duplicate members.
- Dictionaries are used to store data values in key:value pairs.
- Dictionaries are written with curly brackets, and have keys and values:

```
1 thisdict = {
2 "brand": "Ford",
3 "model": "Mustang",
4 "year": 1964
5 }
```

- Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

```
1 thisdict = {
2 "brand": "Ford",
3 "model": "Mustang",
4 "year": 1964
5 }
6 print(thisdict["brand"])
```

- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key.
- Duplicate values will overwrite existing values.
- To determine how many items a dictionary has, use the `len()` function.

```
print(len(thisdict))
```

- The values in dictionary items can be of any data type

```
1 thisdict = {
2 "brand": "Ford",
3 "electric": False,
4 "year": 1964,
5 "colors": ["red", "white", "blue"]
6 }
```

# Conditional branching

```
1 if condition:
2 pass
3 elif condition2:
4 pass
5 else:
6 pass
```

# Loops

Python has two primitive loop commands:

1. while loops
2. for loops

## While loop

- With the `while` loop we can execute a set of statements as long as a condition is true.
- Example: Print i as long as i is less than 6

```
1 i = 1
2 while i < 6:
3 print(i)
4 i += 1
```

- The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.
- With the `break` statement we can stop the loop even if the while condition is true
- With the `continue` statement we can stop the current iteration, and continue with the next.
- With the `else` statement we can run a block of code once when the condition no longer is true.

## For loop

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```

1 fruits = ["apple", "banana", "cherry"]
2 for x in fruits:
3 print(x)

```

- The for loop does not require an indexing variable to set beforehand.
- To loop through a set of code a specified number of times, we can use the range() function.
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3).
- The else keyword in a for loop specifies a block of code to be executed when the loop is finished. A nested loop is a loop inside a loop.
- The “inner loop” will be executed one time for each iteration of the “outer loop”:

```

1 adj = ["red", "big", "tasty"]
2 fruits = ["apple", "banana", "cherry"]

```

```

1 for x in adj:
2 for y in fruits:
3 print(x, y)

```

- for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```

1 for x in [0, 1, 2]:
2 pass

```

---

## Function definition

```
1 def function_name():
2 return
```

---

## Function call

```
function_name()
```

- We need not to specify the return type of the function.
  - Functions by default return `None`
  - We can return any datatype.
- 

## Python Syntax

Python syntax was made for readability, and easy editing. For example, the python language uses a `:` and indented code, while javascript and others generally use `{}` and indented code.

---

## First Program

Lets create a python 3 repl, and call it *Hello World*. Now you have a blank file called *main.py*. Now let us write our first line of code:

```
print('Hello world!')
```

---

*Brian Kernighan actually wrote the first "Hello, World!" program as part of the documentation for the BCPL programming language developed by Martin Richards.*

Now, press the run button, which obviously runs the code. If you are not using repl.it, this will not work. You should research how to run a file with your text editor.

---

## Command Line

If you look to your left at the console where hello world was just printed, you can see a `>`, `>>>`, or `$` depending on what you are using. After the prompt, try typing a line of code.

```
1 Python 3.6.1 (default, Jun 21 2017, 18:48:35)
2 [GCC 4.9.2] on linux
3 Type "help", "copyright", "credits" or "license" for more information.
4 > print('Testing command line')
5 Testing command line
6 > print('Are you sure this works?')
7 Are you sure this works?
8 >
```

The command line allows you to execute single lines of code at a time. It is often used when trying out a new function or method in the language.

---

## New: Comments!

Another cool thing that you can generally do with all languages, are comments. In python, a comment starts with a `#`. The computer ignores all text starting after the `#`.

```
Write some comments!
```

If you have a huge comment, do **not** comment all the 350 lines, just put `'''` before it, and `'''` at the end. Technically, this is not a comment but a string, but the computer still ignores it,

so we will use it.

---

## New: Variables!

Unlike many other languages, there is no `var`, `let`, or `const` to declare a variable in python. You simply go `name = 'value'`.

Remember, there is a difference between integers and strings. *Remember: String = ""*. To convert between these two, you can put an int in a `str()` function, and a string in a `int()` function. There is also a less used one, called a float. Mainly, these are integers with decimals. Change them using the `float()` command.

[https://repl.it/@bgoonz/second-scr?  
lite=true&referrer=https%3A%2F%2Fbryanguner.medium.com](https://repl.it/@bgoonz/second-scr?lite=true&referrer=https%3A%2F%2Fbryanguner.medium.com)

```
1 x = 5
2 x = str(x)
3 b = '5'
4 b = int(b)
5 print('x = ', x, '; b = ', str(b), ';') # => x = 5; b = 5;
```

Instead of using the `,` in the `print` function, you can put a `+` to combine the variables and string.

---

## Operators

There are many operators in python:

- `+`
- `-`
- `/`
- `*` These operators are the same in most languages, and allow for addition, subtraction, division, and multiplication. Now, we can look at a few more complicated ones:

# Python Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+X, -X, ~X	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
		Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, Identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

*simpleops.py*

```
1 x = 4
2 a = x + 1
3 a = x - 1
4 a = x * 2
5 a = x / 2
```

You should already know everything shown above, as it is similar to other languages. If you continue down, you will see more complicated ones.

*complexop.py*

```
1 a += 1
2 a -= 1
3 a *= 2
4 a /= 2
```

The ones above are to edit the current value of the variable.

Sorry to JS users, as there is no `i++;` or anything.

---

## Fun Fact: The python language was named after Monty Python.

If you really want to know about the others, view Py Operators

---

## More Things With Strings

Like the title?

Anyways, a `'` and a `"` both indicate a string, but **do not combine them!**

*quotes.py*

```
1 x = 'hello' # Good
2 x = "hello" # Good
3 x = "hello' # ERRORRR!!!
```

*slicing.py*

---

## String Slicing

You can look at only certain parts of the string by slicing it, using `[num:num]` .

The first number stands for how far in you go from the front, and the second stands for how far in you go from the back.

```
1 x = 'Hello everybody!'
2 x[1] # 'e'
3 x[-1] # '!'
4 x[5] # ' '
5 x[1:] # 'ello everybody!'
6 x[:-1] # 'Hello everybod'
7 x[2:-3] # 'llo everyb'
```

## Methods and Functions

Here is a list of functions/methods we will go over:

- `.strip()`
- `len()`
- `.lower()`
- `.upper()`
- `.replace()`
- `.split()`

## New: `input()`

`input` is a function that gathers input entered from the user in the command line. It takes one optional parameter, which is the users prompt.

*inp.py*

```
1 print('Type something: ')
2 x = input()
3 print('Here is what you said: ', x)
```

---

If you wanted to make it smaller, and look neater to the user, you could do...

*inp2.py*

```
print('Here is what you said: ', input('Type something: '))
```

Running:

*inp.py*

```
1 Type something:
2 Hello World
3 Here is what you said: Hello World
```

*inp2.py*

```
1 Type something: Hello World
2 Here is what you said: Hello World
```

---

## New: Importing Modules

Python has created a lot of functions that are located in other .py files. You need to import these **modules** to gain access to them. You may wonder why python did this. The purpose of separate modules is to make python faster. Instead of storing millions and millions of functions, it only needs a few basic ones. To import a module, you must write

```
input <modulename> . Do not add the .py extension to the file name. In this example , we will be using a python created module named random.
```

*module.py*

```
import random
```

Now, I have access to all functions in the random.py file. To access a specific function in the module, you would do `<module>.<function>`. For example:

*module2.py*

```
1 import random
2 print(random.randint(3,5)) # Prints a random number between 3 and 5
```

*Pro Tip:*

*Do `from random import randint` to not have to do `random.randint()`, just `randint()`*  
*To import all functions from a module, you could do `from random import *`*

## New: Loops!

Loops allow you to repeat code over and over again. This is useful if you want to print Hi with a delay of one second 100 times.

### for Loop

The for loop goes through a list of variables, making a separate variable equal one of the list every time.

Let's say we wanted to create the example above.

*loop.py*

```
1 from time import sleep
2 for i in range(100):
3 print('Hello')
4 sleep(.3)
```

This will print Hello with a .3 second delay 100 times. This is just one way to use it, but it is usually used like this:

*loop2.py*

```
1 import time
2 for number in range(100):
3 print(number)
4 time.sleep(.1)
```

[https://storage.googleapis.com/replit/images/1539649280875\\_37d22e6d49e8e8fbc453631def345387.png](https://storage.googleapis.com/replit/images/1539649280875_37d22e6d49e8e8fbc453631def345387.png)

## while Loop

The while loop runs the code while something stays true. You would put `while <expression>`. Every time the loop runs, it evaluates if the expression is True. If it is, it runs the code, if not it continues outside of the loop. For example:

*while.py*

```
1 while True: # Runs forever
2 print('Hello World!')
```

Or you could do:

*while2.py*

```
1 import random
2 position = '<placeholder>'
3 while position != 1: # will run at least once
4 position = random.randint(1, 10)
5 print(position)
```

## New: if Statement

The if statement allows you to check if something is True. If so, it runs the code, if not, it continues on. It is kind of like a while loop, but it executes **only once**. An if statement is written:

*if.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('num is 3. Hooray!!!')
5 if num > 5:
6 print('Num is greater than 5')
7 if num == 12:
8 print('Num is 12, which means that there is a problem with the python lan
```

Now, you may think that it would be better if you could make it print only one message. Not as many that are True. You can do that with an `elif` statement:

*elif.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('Num is three, this is the only msg you will see.')
5 elif num > 2:
6 print('Num is not three, but is greater than 1')
```

Now, you may wonder how to run code if none work. Well, there is a simple statement called `else:`

*else.py*

```
1 import random
2 num = random.randint(1, 10)
3 if num == 3:
4 print('Num is three, this is the only msg you will see.'
```

```
5 elif num > 2:
6 print('Num is not three, but is greater than 1')
7 else:
8 print('No category')
```

## New: Functions ( def )

So far, you have only seen how to use functions other people have made. Let use the example that you want to print the a random number between 1 and 9, and print different text every time. It is quite tiring to type:

Characters: 389

*nofunc.py*

```
1 import random
2 print(random.randint(1, 9))
3 print('Wow that was interesting.')
4 print(random.randint(1, 9))
5 print('Look at the number above ^')
6 print(random.randint(1, 9))
7 print('All of these have been interesting numbers.')
8 print(random.randint(1, 9))
9 print("these random.randint's are getting annoying to type")
10 print(random.randint(1, 9))
11 print('Hi')
12 print(random.randint(1, 9))
13 print('j')
```

Now with functions, you can seriously lower the amount of characters:

Characters: 254

*functions.py*

```
1 import random
2 def r(t):
```

```
3 print(random.randint(1, 9))
4 print(t)
5 r('Wow that was interesting.')
6 r('Look at the number above ^')
7 r('All of these have been interesting numbers.')
8 r("these random.randint's are getting annoying to type")
9 r('Hi')
10 r('j')
```

---

## Project Based Learning:

The following is a modified version of a tutorial posted By: InvisibleOne

I would cite the original tutorial it's self but at the time of this writing I can no longer find it on his repl.it profile and so the only reference I have are my own notes from following the tutorial when I first found it.

---

### 1. Adventure Story

The first thing you need with an adventure story is a great storyline, something that is exciting and fun. The idea is, that at each pivotal point in the story, you give the player the opportunity to make a choice.

First things first, let's import the stuff that we need, like this:

```
1 import os #very useful for clearing the screen
2 import random
```

Now, we need some variables to hold some of the player data.

```
1 name = input("Name Please: ") #We'll use this to get the name from the user
2 nickname = input("Nickname: ")
```

Ok, now we have the player's name and nickname, let's welcome them to the game

```
print("Hello and welcome " + name)
```

Now for the story. The most important part of all stories is the introduction, so let's print our introduction

```
1 print("Long ago, there was a magical meal known as Summuh and Spich Atip") #We
2 print("It was said that this meal had the power to save lives, restore peace,
```

Now, we'll give the player their first choice

```
1 print("After hiking through the wastelands for a long time, you come to a mass")
2 choice1 = input("[1] Take the bridge [2] Try and jump over")
3 #Now we check to see what the player chose
4 If choice1 == '1':
5 print("You slowly walk across the bridge, it creaks ominously, then suddenly")
6 #The player lost, so now we'll boot them out of the program with the exit co
7 exit()
8 #Then we check to see if they made the other choice, we can do with with else
9 elif choice1 == '2':
10 print("You make the jump! You see a feather hit the bridge, the weight break
11 #Now we can continue the story
12 print("A few more hours of travel and you come to the unclimbable mountain.")
13 choice2 == input("[1] Give up [2] Try and climb the mountain")
14 if choice2 == '1':
15 print("You gave up and lost...")
16 #now we exit them again
17 exit()
18 elif choice2 == '1':
19 print("you continue up the mountain. Climbing is hard, but finally you reach
20 print("Old Man: Hey " + nickname)
21 print("You: How do you know my name!?!")
22 print("Old Man: Because you have a name tag on...")
23 print("You: Oh, well, were is the Summuh and Spich Atip?")
24 print("Old Man: Summuh and Spich Atip? You must mean the Pita Chips and Humr
25 print("You: Pita...chips...humus, what power do those have?")
26 print("Old Man: Pretty simple kid, their organic...")
27 #Now let's clear the screen
28 os.system('clear')
29 print("YOU WON!!!!")
```

---

There you have it, a pretty simple choose your own ending story. You can make it as complex or uncomplex as you like.

---

## 2. TEXT ENCODER

Ever make secret messages as a kid? I used to. Anyways, here's the way you can make a program to encode messages! It's pretty simple. First things first, let's get the message the user wants to encode, we'll use `input()` for that:

```
message = input("Message you would like encoded: ")
```

Now we need to split that string into a list of characters, this part is a bit more complicated.

```
1 #We'll make a function, so we can use it later
2 def split(x):
3 return [char for char in x]
4 #now we'll call this function with our text
5 L_message = message.lower() #This way we can lower any of their input
6 encode = split(l_message)
```

Now we need to convert the characters into code, well do this with a for loop:

```
1 out = []
2 for x in encode:
3 if x == 'a':
4 out.append('1')
5 elif x == 'b':
6 out.append('2')
7 #And we'll continue on though this with each letter of the alphabet
```

---

Once we've encoded the text, we'll print it back for the user

---

```
1 x = ' '.join(out)
2 #this will turn out into a string that we can print
3 print(x)
```

And if you want to decode something, it is this same process but in reverse!

---

### 3. Guess my Number

Number guessing games are fun and pretty simple, all you need are a few loops. To start, we need to import random.

```
import random
```

That is pretty simple. Now we'll make a list with the numbers we want available for the game

```
num_list = [1,2,3,4,5,6,7,8,9,10]
```

Next, we get a random number from the list

```
num = random.choice(num_list)
```

Now, we need to ask the user for input, we'll do this with a while loop

```
1 while True:
2 # We could use guess = input("What do you think my number is? "), but that
3 # would be a string, so we'd have to convert it to an integer
4 # Next, we'll check if that number is equal to the number we picked
5 if guess == num:
6 break #this will remove us from the loop, so we can display the win message
7 else:
```

```
8 print("Nope, that isn't it")
9 #outside our loop, we'll have the win message that is displayed if the player
10 print("You won!")
```

Have fun with this!

---

## 4. Notes

Here is a more advanced project, but still pretty easy. This will be using a txt file to save some notes. The first thing we need to do is to create a txt file in your repl, name it 'notes.txt'

Now, to open a file in python we use `open('filename', type)` The type can be 'r' for read, or 'w' for write. There is another option, but we won't be using that here. Now, the first thing we are going to do is get what the user would like to save:

```
message = input("What would you like to save?")
```

Now we'll open our file and save that text

```
1 o = open('notes.txt', 'w')
2 o.write(message)
3 #this next part is very important, you need to always remember to close your
4 o.close()
```

There we go, now the information is in the file. Next, we'll retrieve it

```
1 read = open('notes.txt', 'r')
2 out = read.read()
3 # now we need to close the file
4 read.close()
5 # and now print what we read
6 print(out)
```

There we go, that's how you can open files and close files with python

---

## 5. Random Dare Generator

Who doesn't love a good dare? Here is a program that can generate random dares. The first thing we'll need to do is as always, import random. Then we'll make some lists of dares

```
1 import random
2 list1 = ['jump on', 'sit on', 'rick roll on', 'stop on', 'swing on']
3 list2 = ['your cat', 'your neighbor', 'a dog', 'a tree', 'a house']
4 list3 = ['your mom', 'your best friend', 'your dad', 'your teacher']
5 #now we'll generate a dare
6 while True:
7 if input() == '': #this will trigger if they hit enter
8 print("I dare you to " + random.choice(list1) + ' ' + random.choice(list2))
```



PythonPracticeGists-1

<https://replit.com/@bgoonz/PythonPracticeGists-1>

## Resources

# Reference Materials

# Appendix

## Interactive Mode

### Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually Control-C or Delete) to the primary or secondary prompt cancels the input and returns to the primary prompt. Typing an interrupt while a command is executing raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

### Executable Python Scripts

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line :

```
#!/usr/bin/env python3.5
```

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash, or pound, character, `'#'`, is used to start a comment in Python.

The script can be given an executable mode, or permission, using the `chmod` command.

```
``` {.sourceCode .shell-session} $ chmod +x myscript.py
```

...

On Windows systems, there is no notion of an "executable mode". The Python installer automatically associates `.py` files with `python.exe` so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in that case, the console window that normally appears is suppressed.

The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like

```
if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read()) . If you want to use the startup file in a script, you must do this explicitly in the script:
```

```
1 import os
2 filename = os.environ.get('PYTHONSTARTUP')
3 if filename and os.path.isfile(filename):
4     with open(filename) as fobj:
5         startup_file = fobj.read()
6     exec(startup_file)
```

The Customization Modules

Python provides two hooks to let you customize it: `sitcustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
||| import site site.getusersitepackages() '/home/user/.local/lib/python3.5/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-s` option to disable the automatic import.

`sitecustomize` works in the same way, but is typically created by an administrator of the computer in the global `site-packages` directory, and is imported before `usercustomize`. See the documentation of the `site` module for more details.

Footnotes

Glossary

Glossary

>>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter. ...

Can refer to:

- The default Python prompt of the interactive shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- The `Ellipsis` built-in constant.

2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See [2to3 - Automated Python 2 to 3 code translation](#).

Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module's `annotation`

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a type hint.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See [variable annotation](#), [function annotation](#), [PEP 484](#) and [PEP 526](#), which describe this functionality.

A value passed to a function (or method) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
1 complex(real=3, imag=5)
2 complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
1 complex(3, 5)
2 complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry, the FAQ question on the difference between arguments and parameters, and [PEP 362](#).asynchronous context manager

An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by [PEP 492](#).asynchronous generator

A function which returns an asynchronous generator iterator. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.asynchronous generator iterator

An object created by a asynchronous generator function.

This is an asynchronous iterator which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator*

iterator effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See [PEP 492](#) and [PEP 525](#).asynchronous iterable

An object, that can be used in an `async for` statement. Must return an asynchronous iterator from its `__aiter__()` method. Introduced by [PEP 492](#).asynchronous iterator

An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an awaitable object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by [PEP 492](#).attribute

A value associated with an object which is referenced by name using dotted expressions. For example, if an object `o` has an attribute `a` it would be referenced as `o.a.awaitable`

An object that can be used in an `await` expression. Can be a coroutine or an object with an `__await__()` method. See also [PEP 492](#).BDFL

Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.binary file

A file object able to read and write bytes-like objects. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also [text file](#) for a file object able to read and write `str` objects.bytes-like object

An object that supports the Buffer Protocol and can export a C-contiguous buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as “read-write bytes-like objects”. Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects (“read-only bytes-like objects”); examples of these include `bytes` and a `memoryview` of a `bytes` object.bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the

same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.`callback`

A subroutine function which is passed as an argument to be executed at some point in the `future.class`

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the `class.class` variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).`coercion`

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.`complex number`

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you’re not aware of a need for them, it’s almost certain you can safely ignore them.`context manager`

An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.`context variable`

A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable.

However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also [PEP 492](#).coroutine function

A function which returns a coroutine object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by [PEP 492](#).CPython

The canonical implementation of the Python programming language, as distributed on python.org. The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
1 def f(...):
2     ...
3 f = staticmethod(f)
4
5 @staticmethod
6 def f(...):
7     ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.descriptor

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see [Implementing Descriptors](#) or the [Descriptor How To Guide](#).

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.dictionary comprehension

A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See [Displays for lists, sets and dictionaries](#).

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See [Dictionary view objects](#).

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

A module written in C or C++, using Python's C API to interact with the core and with user code.`f-string`

String literals prefixed with `'f'` or `'F'` are commonly called "f-strings" which is short for formatted string literals. See also [PEP 498](#).

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

A synonym for `file` object.`finder`

An object that tries to find the loader for a module that is being imported.

Since Python 3.3, there are two types of finder: meta path finders for use with `sys.meta_path`, and path entry finders for use with `sys.path_hooks`.

See [PEP 302](#), [PEP 420](#) and [PEP 451](#) for much more detail.

Mathematical division that rounds down to nearest integer. The floor division operator is `//`.

For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by

float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*.

See [PEP 238](#).function

A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the Function definitions section.function annotation

An annotation of a function parameter or return value.

Function annotations are usually used for type hints: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
1 def sum_two_numbers(a: int, b: int) -> int:
2     return a + b
```

Function annotation syntax is explained in section Function definitions.

See variable annotation and [PEP 484](#), which describe this functionality.__future__

A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:>>>

```
1 >>> import __future__
2 >>> __future__.division
3 _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.generator

A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An object created by a generator function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

An expression that returns an iterator. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:>>>

```
1 >>> sum(i*i for i in range(10))          # sum of squares 0, 1, 4, ... 81
2 285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the single dispatch glossary entry, the `functools.singledispatch()` decorator, and [PEP 443](#).generic type

A type that can be parameterized; typically a container like `list`. Used for type hints and annotations.

See [PEP 483](#) for more details, and `typing` or generic alias type for its uses.

See [global interpreter lock](#).

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.`hash-based pyc`

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See [Cached bytecode invalidation](#).

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.
IDLE

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.`import path`

A list of locations (or path entries) that are searched by the path based finder for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

The process by which Python code in one module is made available to Python code in another module.

An object that both finds and loads a module; both a finder and loader object.

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also `interactive.interpreter.shutdown`

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the garbage collector. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, file objects, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements Sequence semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function

`iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also `iterator`, `sequence`, and `generator.iterator`

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in `Iterator Types.key function`

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the `Sorting HOW TO` for examples of how to create and use key functions.`keyword argument`

See `argument.lambda`

An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is

```
lambda [parameters]: expression LBYL
```

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code,

```
if key in mapping: return mapping[key]
```

 can fail if another thread removes `key` from `mapping` after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.`.list`

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.`.list` comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.`.loader`

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a finder. See [PEP 302](#) for details and `importlib.abc.Loader` for an abstract base class.`.magic` method

An informal synonym for `special method.mapping`

A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.`.meta` path finder

A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from path entry finders.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.`.metaclass`

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in [Metaclasses.method](#)

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See [function](#) and [nested scope.method resolution order](#)

Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

See also [package.module spec](#)

A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec.MRO`

See [method resolution order.mutable](#)

Mutable objects can change their value but keep their `id()`. See also [immutable.named tuple](#)

The term “named tuple” applies to any type or class that inherits from `tuple` and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info :>>>`

```
1 >>> sys.float_info[1]                      # indexed access
2 1024
3 >>> sys.float_info.max_exp                 # named field access
4 1024
5 >>> isinstance(sys.float_info, tuple)      # kind of tuple
6 True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.`namespace`

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces.

Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.`namespace` package

A **PEP 420** package which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a regular package because they have no `__init__.py` file.

See also `module.nested scope`

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.`new-style class`

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.`object`

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.package

A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also [regular package](#) and [namespace package](#).parameter

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example `foo` and `bar` in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example `posonly1` and `posonly2` in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example `kw_only1` and `kw_only2` in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters).

Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the Function definitions section, and [PEP 362](#).path entry

A single location on the import path which the path based finder consults to find modules for importing.path entry finder

A finder returned by a callable on `sys.path_hooks` (i.e. a path entry hook) which knows how to locate modules given a path entry.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.path entry hook

A callable on the `sys.path_hook` list which returns a path entry finder if it knows how to find modules on a specific path entry.path based finder

One of the default meta path finders which searches an import path for modules.path-like object

An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by

calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.positional argument

See argument.provisional API

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a “solution of last resort” - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.provisional package

See provisional API.Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
1 for i in range(len(food)):
2     print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
1 for piece in food:
2     print(piece)
```

qualified name

A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in [PEP 3155](#). For top-level functions and classes, the qualified name is the same as the object’s name:>>>

```
1 >>> class C:
2     ...     class D:
3         ...         def meth(self):
4             ...             pass
5 ...
6 >>> C.__qualname__
7 'C'
8 >>> C.D.__qualname__
9 'C.D'
10 >>> C.D.meth.__qualname__
11 'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text` :>>>

```
1 >>> import email.mime.text
2 >>> email.mime.text.__name__
3 'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

A traditional package, such as a directory containing an `__init__.py` file.

See also [namespace package](#).`__slots__`

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See [Displays for lists, sets and dictionaries](#).single dispatch

A form of generic function dispatch where the implementation is chosen based on the type of a single argument.

An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in `Special method names`.

A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`.

A codec which encodes Unicode strings to bytes.

A file object able to read and write `str` objects. Often, a text file actually accesses a byte-oriented data stream and handles the text encoding automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also `binary file` for a file object able to read and write bytes-like objects.

A string which is bound by three instances of either a quotation mark (“) or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying type hints. For example:

```
1 def remove_gray_shades(  
2     colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:  
3     pass
```

could be made more readable like this:

```
1 Color = tuple[int, int, int]
2
3 def remove_gray_shades(colors: list[Color]) -> list[Color]:
4     pass
```

See `typing` and **PEP 484**, which describe this functionality.type hint

An annotation that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.variable annotation

An annotation of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
1 class C:
2     field: 'annotation'
```

Variable annotations are usually used for type hints: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section [Annotated assignment statements](#).

See function annotation, [PEP 484](#) and [PEP 526](#), which describe this functionality.

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also [venv](#) .virtual machine

A computer defined entirely in software. Python's virtual machine executes the bytecode emitted by the bytecode compiler.

Zen of Python
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “`import this`” at the interactive prompt.

Binary Distribution

A specific kind of Built Distribution that contains compiled extensions.

Built Distribution
A Distribution format containing files and metadata that only need to be moved to the correct location on the target system, to be installed. Wheel is such a format, whereas distutil's Source Distribution is not, in that it requires a build step before it can be installed. This format does not imply that Python files have to be precompiled (Wheel intentionally does not include compiled Python files).

Distribution Package
A versioned archive file that contains Python packages, modules, and other resource files that are used to distribute a Release. The archive file is what an end-user will download from the internet and install.

A distribution package is more commonly referred to with the single words “package” or “distribution”, but this guide may use the expanded term when more clarity is needed to prevent confusion with an Import Package (which is also commonly called a “package”) or another

kind of distribution (e.g. a Linux distribution or the Python language distribution), which are often referred to with the single term “distribution”.Egg

A Built Distribution format introduced by `setuptools`, which is being replaced by `Wheel`. For details, see ` [The Internal Structure of Python Eggs and Python EggsExtension Module](#)

A Module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions.Known Good Set (KGS)

A set of distributions at specified versions which are compatible with each other. Typically a test suite will be run which passes all tests before a specific set of packages is declared a known good set. This term is commonly used by frameworks and toolkits which are comprised of multiple individual distributions.Import Package

A **Python module** which can contain other modules or recursively, other packages.

An **import package** is more commonly referred to with the single word “package”, but this guide will use the expanded term when more clarity is needed to prevent confusion with a Distribution Package which is also commonly called a “package”.Module

The basic unit of code reusability in Python, existing in one of two types: Pure Module, or Extension Module.Package Index

A repository of distributions with a web interface to automate package discovery and consumption.Per Project Index

A private or other non-canonical Package Index indicated by a specific Project as the index preferred or required to resolve dependencies of that project.Project

A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof that is intended to be packaged into a Distribution.

Since most projects create Distributions using either **PEP 518** `build-system` , `distutils` or `setuptools`, another practical way to define projects currently is something that contains a `pyproject.toml`, `setup.py`, or `setup.cfg` file at the root of the project source directory.

Python projects must have unique names, which are registered on PyPI. Each project will then contain one or more Releases, and each release may comprise one or more distributions.

Note that there is a strong convention to name a project after the name of the package that is imported to run that project. However, this doesn't have to hold true. It's possible to install a distribution from the project 'foo' and have it provide a package importable only as 'bar'.Pure Module

A Module written in Python and contained in a single `.py` file (and possibly associated `.pyc` and/or `.pyo` files).Python Packaging Authority (PyPA)

PyPA is a working group that maintains many of the relevant projects in Python packaging. They maintain a site at <https://www.pypa.io>, host projects on GitHub and Bitbucket, and discuss issues on the distutils-sig mailing list and the Python Discourse forum.Python Package Index (PyPI)

PyPI is the default Package Index for the Python community. It is open to all Python developers to consume and distribute their distributions.pypi.org

pypi.org is the domain name for the Python Package Index (PyPI). It replaced the legacy index domain name, `pypi.python.org`, in 2017. It is powered by Warehouse.`pyproject.toml`

The tool-agnostic Project specification file. Defined in **PEP 518**.Release

A snapshot of a Project at a particular point in time, denoted by a version identifier.

Making a release may entail the publishing of multiple Distributions. For example, if version 1.0 of a project was released, it could be available in both a source distribution format and a Windows installer distribution format.Requirement

A specification for a package to be installed. pip, the PYPA recommended installer, allows various forms of specification that can all be considered a "requirement". For more information, see the pip install reference.Requirement Specifier

A format used by pip to install packages from a Package Index. For an EBNF diagram of the format, see the `pkg_resources.Requirement` entry in the setuptools docs. For example, "foo>=1.3" is a requirement specifier, where "foo" is the project name, and the ">=1.3" portion is the Version SpecifierRequirements File

A file containing a list of Requirements that can be installed using pip. For more information, see the pip docs on Requirements Files.`setup.py``setup.cfg`

The project specification files for `distutils` and `setuptools`. See also `pyproject.toml`.Source Archive

An archive containing the raw source code for a Release, prior to creation of a Source Distribution or Built Distribution.Source Distribution (or “`sdist`”)

A distribution format (usually generated using `python setup.py sdist`) that provides metadata and the essential source files needed for installing by a tool like pip, or for generating a Built Distribution.System Package

A package provided in a format native to the operating system, e.g. an rpm or dpkg file.Version Specifier

The version component of a Requirement Specifier. For example, the “`>=1.3`” portion of “`foo>=1.3`”. **PEP 440** contains a **full specification** of the specifiers that Python packaging currently supports. Support for PEP440 was implemented in `setuptools` v8.0 and pip v6.0.Virtual Environment¶

An isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide. For more information, see the section on Creating Virtual Environments.Wheel

A Built Distribution format introduced by **PEP 427**, which is intended to replace the Egg format. Wheel is currently supported by pip.Working Set

A collection of distributions available for importing. These are the distributions that are on the `sys.path` variable. At most, one Distribution for a project is possible in a working set.

Errors

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
while True print('Hello world') File "", line 1 while True print('Hello world') ^ SyntaxError:  
invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function `print`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
10 -(1/0) Traceback (most recent call last): File "", line 1, in ZeroDivisionError: division by  
zero 4 + spam_3 Traceback (most recent call last): File "", line 1, in NameError: name  
'spam' is not defined '2' + 2 Traceback (most recent call last): File "", line 1, in TypeError:  
can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`,

NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

`bltin-exceptions` lists the built-in exceptions and their meanings.

Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception. :

```
while True: ... try: ... x = int(input("Please enter a number: ")) ... break ... except ValueError:  
... print("Oops! That was no valid number. Try again...") ...
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the `except` clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the `except` clause, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
1 ... except (RuntimeError, TypeError, NameError):  
2     pass
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
1 class B(Exception):  
2     pass  
3  
4 class C(B):  
5     pass  
6  
7 class D(C):  
8     pass  
9  
10 for cls in [B, C, D]:  
11     try:  
12         raise cls()  
13     except D:  
14         print("D")  
15     except C:  
16         print("C")  
17     except B:  
18         print("B")
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
1 import sys
2
3 try:
4     f = open('myfile.txt')
5     s = f.readline()
6     i = int(s.strip())
7 except OSError as err:
8     print("OS error: {0}".format(err))
9 except ValueError:
10    print("Could not convert data to an integer.")
11 except:
12    print("Unexpected error:", sys.exc_info()[0])
13 raise
```

The try ... except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
1 for arg in sys.argv[1:]:
2     try:
3         f = open(arg, 'r')
4     except OSError:
5         print('cannot open', arg)
6     else:
7         print(arg, 'has', len(f.readlines()), 'lines')
8         f.close()
```

The use of the !else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the !try ... !except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__` so the arguments can be printed directly without having to reference `.args`. One may also instantiate an exception first before raising it and add any attributes to it as desired. :

```
try: ... raise Exception('spam', 'eggs') ... except Exception as inst: ... print(type(inst)) # the exception instance ... print(inst.args) # arguments stored in .args ... print(inst) # str allows args to be printed directly, ... # but may be overridden in exception subclasses ... x, y = inst.args # unpack args ... print('x =', x) ... print('y =', y) ... ('spam', 'eggs') ('spam', 'eggs') x = spam y = eggs
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
def this_fails(): ... x = 1/0 ... try: ... this_fails() ... except ZeroDivisionError as err: ... print('Handling run-time error:', err) ... Handling run-time error: division by zero
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
raise NameError('HiThere') Traceback (most recent call last): File "", line 1, in NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try: ... raise NameError('HiThere') ... except NameError: ... print('An exception flew by!') ... raise ... An exception flew by! Traceback (most recent call last): File "", line 2, in
```

```
NameError: HiThere
```

Exception Chaining

The `raise` statement allows an optional `from` which enables chaining exceptions. For example:

```
1 # exc must be exception instance or None.  
2 raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func(): ... raise IOError ... try: ... func() ... except IOError as exc: ... raise  
RuntimeError('Failed to open database') from exc ... Traceback (most recent call last):  
File "", line 2, in File "", line 2, in func OSError The above exception was the direct cause of  
the following exception: Traceback (most recent call last): File "", line 4, in RuntimeError:  
Failed to open database
```

Exception chaining happens automatically when an exception is raised inside an `except` or `finally` section. Exception chaining can be disabled by using `from None` idiom:

```
>>> try: ... open('database.sqlite') ... except OSError: ... raise RuntimeError from None ...  
Traceback (most recent call last): File "\", line 4, in RuntimeError
```

For more information about chaining mechanics, see `bltin-exceptions`.

User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see `tut-classes` for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to

be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
1  class Error(Exception):
2      """Base class for exceptions in this module."""
3      pass
4
5  class InputError(Error):
6      """Exception raised for errors in the input.
7
8      Attributes:
9          expression -- input expression in which the error occurred
10         message -- explanation of the error
11     """
12
13     def __init__(self, expression, message):
14         self.expression = expression
15         self.message = message
16
17 class TransitionError(Error):
18     """Raised when an operation attempts a state transition that's not
19     allowed.
20
21     Attributes:
22         previous -- state at beginning of transition
23         next -- attempted new state
24         message -- explanation of why the specific transition is not allowed
25     """
26
27     def __init__(self, previous, next, message):
28         self.previous = previous
29         self.next = next
30         self.message = message
```

Most exceptions are defined with names that end in "Error", similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter tut-classes.

Defining Clean-up Actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
| | | try: ... raise KeyboardInterrupt ... finally: ... print('Goodbye, world!') ... Goodbye, world!
| | | Traceback (most recent call last): File "", line 2, in KeyboardInterrupt
```

If a finally clause is present, the !finally clause will execute as the last task before the try statement completes. The !finally clause runs whether or not the !try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the !try clause, the exception may be handled by an except clause. If the exception is not handled by an !except clause, the exception is re-raised after the !finally clause has been executed.
- An exception could occur during execution of an !except or !else clause. Again, the exception is re-raised after the !finally clause has been executed.
- If the !finally clause executes a break, continue or return statement, exceptions are not re-raised.
- If the !try statement reaches a break, continue or return statement, the !finally clause will execute just prior to the !break, !continue or !return statement's execution.
- If a !finally clause includes a !return statement, the returned value will be the one from the !finally clause's !return statement, not the value from the !try clause's !return statement.

For example:

```
| | | def bool_return(): ... try: ... return True ... finally: ... return False ... bool_return() False
```

A more complicated example:

```
| | | def divide(x, y): ... try: ... result = x / y ... except ZeroDivisionError: ... print("division by zero!") ... else: ... print("result is", result) ... finally: ... print("executing finally clause") ...
| | | divide(2, 1) result is 2.0 executing finally clause divide(2, 0) division by zero! executing finally clause divide("2", "1") executing finally clause Traceback (most recent call last):
| | | File "", line 1, in File "", line 3, in divide TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the finally clause is executed in any event. The TypeError raised by dividing two strings is not handled by the except clause and therefore re-raised after the !finally clause has been executed.

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen. :

```
1 for line in open("myfile.txt"):
2     print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly. :

```
1 with open("myfile.txt") as f:
2     for line in f:
3         print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

Python Keywords

In the Beginning were Python's Keywords

Let's face it, learning to *write* computer programs is *hard*. Doing so involves picking up a lot of specialized skills and jargon, most of which is unfamiliar, and it also means cultivating a mental approach to the world that, at first, might seem downright *alien*. It's a long journey, but fluency starts with learning how to *read* computer programs, and that *really* shouldn't be so hard.

Programs must be written for people to read, and only incidentally for machines to execute. Harold Abelson

One of the defining traits of the Python programming language is that it's *designed* to be readable by humans. For the most part it fulfills that promise, however, after several years of helping others understand Python, I've come to realize that there's an important caveat to that statement; it's really only true for someone with a *significant* and *specialized* English-language vocabulary.

The truth is you need enough mastery of the "rules" of English grammar and wordplay to recognize and understand, at a glance, words *written* in English that *are not* actually words you'll find in an English dictionary. Nonstandard terms like **def** (a *contraction*¹), **elif** (a *portmanteau*) and **nonlocal** (a *neologism*) all abound, so even for those with quite advanced *native* English fluency the task of learning Python is very much like trying to learn a foreign language that's *slightly* related to their native tongue. For those without that fluency learning Python is like learning a foreign language *inside* a foreign language, and is therefore *far* more difficult.

To make that task a little easier I'm going to try, in this post and the ones that follow, to shed some light on the meaning of – and a *little* of the etymological history behind – the fundamental units of Python fluency. In this first part we will start with the most basic of those units, Python's 35 *reserved keywords*².

That's right; the core vocabulary of Python you actually *need* to know to start to do meaningful work is just 35 keywords. It's not the smallest language, but it's *far* from the largest, and just compare it to the roughly 10,000 words required to achieve basic *native* fluency in a non-programming language.

First, Some Conventions

Python is what is known as a *statement*-oriented language; but what *is* a statement? Well, for the purposes of this article we're just going to say that in Python a statement is a *single* line of code that does *something*. *What* it does, specifically, depends on the building blocks of that statement.

But what are those building blocks? Well, let's define them quickly and *very* roughly, since we'll go into more detail about them in later posts. I'll use UPPERCASE letters to make it easier to visually distinguish these abstract forms from the specific instances we'll talk about later.**KEYWORD**A reserved *word* the meaning of which *cannot* be changed by the user. We will visit all 35 of these in the next section of this article.**OPERATOR**A reserved *symbol* that indicates an action to be performed. For example, `=` is the *assignment* OPERATOR and `+` is the *addition* OPERATOR. There are quite a few others, but we'll save them for the next post. A small number of KEYWORDS behave like OPERATORS, and I'll point those out below.

These are both provided by Python and you can't directly change their meaning, which means that they're somewhat inflexible. To do most work you'll need something more flexible, which is why Python gives you the ability to represent *anything*.**OBJECT**An individual *thing* you can interact with. Unlike KEYWORDS and OPERATORS, you can directly manipulate these, though the degree to which you can manipulate them depends on what *type* of OBJECT they are. You can also use KEYWORDS to define entirely new *types*, which makes them a very expressive way of building new *things* of your own. So expressive, in fact, that practically speaking *everything* you interact with in Python will be an OBJECT.

That can be a bit abstract and hard to wrap your head around at this point, though. For now just know that OBJECTs tend to fall into three main categories.**VALUE**An OBJECT that represents a single, concrete *thing*; for the purposes of this discussion what that thing actually *is* is irrelevant, but as an example, `4` is a VALUE of the **int** (short for *integer*) type and `hello` is a VALUE of the **str** (short for *string*) type. These are both examples of *primitive* types, which have a single meaningful value, but there are also *composite* types for describing things the meaning of which is defined by more than one *attribute*. A real-world example would be a rectangle, which cannot be defined without both height and width. As you'll see below three special KEYWORDS all behave like VALUEs, though as before you cannot change their meaning.**COLLECTION**An OBJECT that groups together or contains other OBJECTs; there are *many* different types of COLLECTIONs in Python, but for the moment all we care about is that a COLLECTION *contains* zero or more OBJECTs. For example the statement `[2, 3, 4]` creates a COLLECTION of the type **list** that holds three VALUEs inside of it. A COLLECTION can contain

any OBJECT, so you can *nest* a COLLECTION inside another COLLECTION. CALLABLE An OBJECT that represents some action to perform: it performs that action when you call it with some number of *arguments* then it *returns* (or gives back) an OBJECT. For instance **sum** is a CALLABLE and when we call it using `sum([2, 3, 4])` it gives us back the VALUE `9`. There are several different kinds of CALLABLE, and we'll touch on them in more detail below.

It wouldn't be very efficient to type out the same OBJECT every time you needed to refer to it, though. It's often very helpful to be able to refer to things *indirectly*. NAME Any word that *is not* a KEYWORD, and that is used as an *alias* to *refer to* some specific OBJECT. Unlike a KEYWORD the meaning of a NAME *may* change over the course of a program, which is why these are often – if a little incorrectly – thought of as *variables*. There are several ways to create new NAMES (and one to destroy them), as we'll see below, but as a simple example in `number = 2` the *assignment* OPERATOR `=` creates the NAME **number** and assigns it to refer to the VALUE **2**. When later that is followed by `number += 2`, however, the *augmented assignment* OPERATOR `+=` will re-assign **number** to refer to **4**.

Now we've got all the simple building blocks defined and we can start organizing them into composite structures. EXPRESSION Any composite form of one or more of the above that can be *evaluated* to an OBJECT. For example, `4`, `2 + 2`, `1 * 3 + 1` and `sum([1, 1, 1, 1])` are all EXPRESSIONs that evaluate to `4`. The EXPRESSION represents the smallest discrete unit of work in Python. STATEMENT Any single line of code that is composed of at least one of the above. These can get quite complex, but to *do* anything they'll usually need to include KEYWORDS and/or OPERATORS plus EXPRESSIONs. You've already met a *useful* STATEMENT in `number = 2`. If you read each STATEMENT in a program out in turn you can track the program as it does its work.

That covers any given *line* of code, but there are also a couple of higher level structures we need to define for the moment:

- BLOCK At least two STATEMENTS that are bound together; the first STATEMENT will end in a `:` character and indicates the start of the BLOCK. The second and all further STATEMENTS inside that BLOCK will be indented further right than the initial STATEMENT, to indicate that they *belong* to the same BLOCK. The last such indented STATEMENT represents the end of the BLOCK.
- MODULE A single Python .py file; it's composed of some number of STATEMENTS. All Python programs are comprised of at least one MODULE. As you'll see below we write all of our functionality inside MODULEs, and we use KEYWORDS and NAMES to *import* functionality from other MODULEs.

There are many other concepts you'll need to become familiar with, but with these building blocks we can investigate all 35 words in Python's relatively small vocabulary, and thus

understand the skeleton of any Python program.

On to the Keywords

One That Does Nothing

`pass`docsA placeholder; technically known as Python's *null operation*, `pass` does nothing whatsoever; it exists solely to allow you to write a syntactically valid BLOCK.

The general meaning here comes from a Middle English verb borrowed via Old French from the Latin *passus* that implies "*to move by [a place, without stopping]*". More specifically the meaning in Python is borrowed from its use in sequential-play card games such as Bridge, where if you do not wish to do anything on your turn you `pass` control of the game to the next player, doing nothing.

The need for this is simple; once you begin a BLOCK in Python it *must* contain *at least one* indented STATEMENT to be considered valid syntax. The `pass` statement exists to allow you to write a valid BLOCK structure *before* you're ready to start writing meaningful statements.

```
1 STATEMENT:  
2     pass
```

Because it's mainly used early on when building the rough structure of a program you'll rarely, if ever, see `pass` in working code, but it's good to know it exists.

Three That Are Objects

The next three keywords are specialized because they each behave like a primitive VALUE. This means they can be assigned to a NAME, kept within a COLLECTION, and can be the result of evaluating an EXPRESSION. They're also the only keywords that start with a capital letter, which makes them easy to distinguish.

boolean values

These two are used in most, if not all, programs, and are essential whenever performing Boolean logic. `True` is a indicator of logical *truth* and the opposite of `False`; behaves like the integer `1`, except that it will always be displayed as `True`.

From the Old English adjective ***triewe***, which has German roots; the general meaning is of being “worthy of trust” and “consistent with fact”. In logic, though, the specific meaning is really just “that which is not false”, and in computer programming it’s usually a proxy for the binary digit `1`. `False` is a indicator of logical *untruth* and the opposite of `True`; behaves like the integer `0`, except that it will always be displayed as `False`.

From late Old English, borrowed via the Old French ***faus*** from the Latin ***falsus***, the general meaning is of “fake, incorrect, mistaken, or deceitful”. In logic the meaning is not so sinister, it just means “that which is not true”, and in computer programming it’s usually a proxy for the binary digit `0`.

In some lower-level languages you would probably just use `1` and `0` for these, but in Python they’ve been given special keywords to make it obvious that you’re working with the *logical* meaning instead of the *numerical* one.

It’s important to understand that in Python *every* OBJECT (hence ever VALUE and COLLECTION, and therefore every EXPRESSION), has a *logical* value, in that it’s considered to be logically equivalent to either `True` or `False`. Testing the state of that logical value is known as truth-value testing, and as you’ll see below keywords like `and`, `or`, and `if` all rely on truth-value testing for their operation.

I will go into deeper detail about the specifics of truth-value testing in later articles, but for now you just need to know that most things are considered `True` by default, except for “no value” VALUES like `0`, `None`, and `False` itself, as well as “no content” COLLECTIONs like `[]`.

the null value

This is most commonly used to represent the absence of any other VALUE. `None` is a special name for nothing whatsoever; technically Python’s *null object*, it’s considered equivalent to `False` for truth-value testing, but essentially represents no value at all. Is very commonly used in Python, and will always appear as `None`.

A Middle English pronoun from the Old English ***nan***, meaning “not one” or “not any”. The meaning in programming, however, relates more to ***null***, which means “not [any] thing” or just

“nothing”. Python chose to use ***none*** because it’s a more commonly familiar word. It also helps to distinguish it from the use of the special value **NULL** in the C programming language, which has a similar meaning but behaves in a *very* different way.

The notion of making *something* that explicitly represents *nothing* might seem a little odd, at first, but the need for **None** becomes obvious when you start building *useful* code.

Three for Making Decisions

Being able to tell if something is considered **True** or **False** isn’t very useful unless you have the means to take different actions based on that knowledge. For that *most* programming languages have some notion of *conditional operations*. In Python there are three keywords dedicated to conditional tasks.**if**docsStarts a conditional BLOCK by checking the *truth-value* of the EXPRESSION that follows it; the STATEMENT(s) indented underneath the **if** will be executed only if the EXPRESSION is considered **True**.

A Middle English conjunction from the Old English ***gif*** which means, “in the event” or “whether” or, oddly, just “***if***”. Has many Scandinavian/Germanic relatives, and possibly arrives via an Old Norse term for “doubt, hesitation”. The general use is to make one word or phrase conditional on another being true, as in “*if it is raining, open your umbrella*”. The sense in computing is more formal but essentially the same; “*if [this condition is true], then [do some action]*”.
elif Optionally *continues* a conditional by adding another BLOCK; if present it *must* follow either the initial **if** or another **elif**. Behaves exactly like an **if**, except that its conditional EXPRESSION will only be evaluated when no previous **if** / **elif** STATEMENT has evaluated as **True**.

Not a proper English word, but instead a portmanteau that contracts ***else*** and ***if*** into a single artificial word, ***elif***. Together it means “otherwise, if” or “as an alternative, if”, both of which imply that the action controlled by the ***elif*** is contingent on the outcome of some previous test or tests. So, in computing, “[***else*** [after checking some prior condition] ***if***[this other condition is True], then [do some other action]]”.
else Optionally *terminates* a conditional by adding a final BLOCK; if present it *must* follow the last **if** / **elif** in the BLOCK. If no previous **if** / **elif** STATEMENT evaluated to **True** then the indented STATEMENT(s) below **else** will be run. Can also be used to terminate blocks started with other KEYWORDS; see **for**, **while**, and **try** below.

An adverb from the Old English ***elles***, meaning “[to do] instead of [some other action]” or “as an alternative”, or just “otherwise”. In computing it means “[check if any prior condition is true] ***else***

[perform some final action]". It is used to take some default or fallback action when no better, more specific action should be taken.

Conditionals are key to a lot of Python programming, and are needed to better explain some of the keywords that follow, so I'll provide a few examples of how they work.

Sometimes you only want to take *any* action if some condition is met; this is the simplest form:

```
1 if EXPRESSION:  
2     STATEMENT
```

Many situations are binary, though, and so you'll always want to take some fallback action if the condition is *not* met:

```
1 if EXPRESSION:  
2     STATEMENT_A  
3 else:  
4     STATEMENT_B
```

In complex cases you may need to have any number of alternative actions based on mutually exclusive conditions, as well as a fallback:

```
1 if EXPRESSION_A:  
2     STATEMENT_A  
3 elif EXPRESSION_B:  
4     STATEMENT_B  
5 else:  
6     STATEMENT_C
```

For the middle, "either/or" case there's another form you will sometimes see, known as the *ternary operator* form. This is useful mainly because, unlike a standard `if` conditional, it behaves like an EXPRESSION, and the value it evaluates to can be directly assigned to a NAME:

```
NAME = STATEMENT_A if EXPRESSION else STATEMENT_B
```

Which is a much shorter way of writing:

```
1 if EXPRESSION:  
2     NAME = STATEMENT_A  
3 else:  
4     NAME = STATEMENT_B
```

We'll find this useful when we look at the OPERATOR-like keywords below.

Five That Are Operators

The next five keywords all behave like an OPERATOR to denote actions performed on OBJECTs and/or EXPRESSIONs.

boolean logic operators

These are used for making meaningful comparisons between things based on their *truth-value*.
`not`docsA *unary* OPERATOR that inverts the *truth-value* of whatever follows it, as in
`not EXPRESSION`. Can be used to invert the meaning of another KEYWORD; see `is not` and
`not in` below.

A Middle English adverb from the Old English *nawiht*, implying “nothing” or “zero”. The general meaning today negates (or flips) the meaning of the word or phrase that follows it. Compare “I do have apples” to “I do **not** have apples”. In programming, however, the specific meaning comes from logical negation, and thus **not** negates true to false, and vice versa.

This is Python's *Boolean negation* OPERATOR, used whenever you need the opposite of the truth-value of a thing. It is *unary*, which means that it acts on whatever is to its immediate right.

The usage of `not` is straightforward:

```
not EXPRESSION
```

If `EXPRESSION` is considered `True` then `not EXPRESSION` evaluates to `False` and otherwise it evaluates to `True`.

That can be easier to understand if you think of `not` as working like the following *ternary if*:

```
False if EXPRESSION else True
```

anddocsA *binary OPERATOR* that checks the *truth-value* of two things, evaluating to the thing on the left if it tested `False`, otherwise to the thing on the right.

An ancient Old English conjunction with Germanic roots vaguely meaning “thereupon” or “next [to]” and used to combine two words or phrases, as in “coffee **and** tea”. The meaning in Python, however, comes entirely from logical conjunction, and implies that either *both* things it combines are true or the whole combination is false.

This is Python’s *Boolean conjunction OPERATOR*; used whenever you need to test if *both* sides of the `and` are considered `True`. It is a *short-circuiting* operation; if the left-hand `EXPRESSION` is considered `False` the entire operation is considered `False` and the right-hand side will never be evaluated at all. And unlike `not` it does not necessarily evaluate to either `True` or `False`, evaluating instead to either the left side (if considered `False`) or the right side.

Thus the usage:

```
EXPRESSION_A and EXPRESSION_B
```

Can be thought of as working like the following *ternary if*:

```
EXPRESSION_B if EXPRESSION_A else EXPRESSION_A
```

Which is why you may one day find yourself surprised to find that `True` and `1` evaluates to `1` while `1` and `True` evaluates to `True`.
A *binary OPERATOR* that checks the *truth-value* of two things, evaluating to the thing on the left if it tested `True`, otherwise to the thing on the right.

Derived from the Old English conjunction *obbe*, meaning “either”, and implying that either of the ideas conjoined are acceptable, as in “coffee *or* tea”. In Python the meaning comes from logical disjunction, and implies that either *one* of the things it combines is true or the whole combination is false.

This is Python’s *Boolean disjunction OPERATOR*; used whenever you need to test if *either* side of the `or` is considered `True`. It is a *short-circuiting* operation; if the left-hand EXPRESSION is considered `True` the entire operation is considered `True` and the right-hand side will never be evaluated at all. Also, unlike `not` it does not necessarily evaluate to either `True` or `False`, evaluating instead to either the left side (if considered `True`) or the right side.

Thus the usage:

```
EXPRESSION_A or EXPRESSION_B
```

Can be thought of as working like the following *ternary if* :

```
EXPRESSION_A if EXPRESSION_A else EXPRESSION_B
```

This subtlety can catch you out when you find that `True or 1` evaluates to `True` but `1 or True` evaluates to `1`.

identity checking operator(s)

A *binary OPERATOR* that tests if the OBJECT on the left has the same identity as the OBJECT on the right and then evaluates to either `True` or `False`. Can be inverted by `not` to become the `is not` operator.

An Old English verb from the Germanic stem ***es-**; it's the third person singular present indicative form of the word ***be***, so it generally means "to be [a thing]". In Python its meaning is specific to *identity*, and implies something more like "to be [some unique thing]."

The usage is straightforward:

```
EXPRESSION_A is EXPRESSION_B
```

And the usage with `not` is:

```
EXPRESSION_A is not EXPRESSION_B
```

The notion of *identity* is a little abstract, but think of it like this: Tom and Bob are twins, they're the same height, age, and weight, and they share the same birthday, but they *do not* have the same identity. Tom is Tom, and Bob is Bob, but Tom is not Bob.

In most implementations of Python the identity of an OBJECT can be thought of as its unique address in memory, and since *every* OBJECT you work with will have its own unique address, `is` is usually only useful for telling if a NAME refers to a specific OBJECT:

```
1 NAME is OBJECT
2 NAME is not OBJECT
```

Or for testing if two different NAMES refer to the same OBJECT in memory.

```
1 NAME is OTHER_NAME
2 NAME is not OTHER_NAME
```

There *are* however some special cases: for instance, `True`, `False`, and `None` are all singletons in memory, meaning there is only *ever* one copy of `True` in *any* Python program. For the most part this is just a space-saving detail you don't need to worry about, but it explains

why in Python we use `VALUE == True` and `not VALUE is True` when checking if something is considered *equivalent* to `True`. Testing *identity* is *not* the same as testing *value*.

For this reason the uses of `is` are limited and specific, which is why you'll only rarely see `is` and `is not` used in practice.

membership testing operator(s)

indocsA *binary OPERATOR* that tests if the OBJECT on the left is a member of the COLLECTION on the right and then evaluates to either `True` or `False`. Also known as Python's *inclusion operator*. Can be inverted by `not` to become the `not in` *exclusion operator*. Also used with `for`, see below.

A Middle English merger of the Old English words *in*, meaning "among", and *inne*, meaning "within" or "inside". The merged word has many usages and meanings, but the general sense here is from the prepositional form, which implies that some thing is contained within or inside some larger thing, as in "a page *in* a book" or "a book *in* a library". In Python it is specifically used for *membership testing*, when checking if an item is contained within a group of items.

The usual usage of `in` is to test if an OBJECT is a member of a specific CONTAINER:

OBJECT `in` CONTAINER

Or to test if OBJECT is `not` a member of CONTAINER:

OBJECT `not in` CONTAINER

It can be tempting to think you can use `is` with `in`, but that's invalid syntax. It helps to remember that since `is`, `is not`, `in`, and `not in` are all binary OPERATORS they *must* have either an OBJECT or EXPRESSION on either side, *not* another KEYWORD.

Four Used to Loop

The above keywords give you everything you need to perform simple decision making and to take basic actions, but they're useless whenever you need to do something *repeatedly*; that's where *looping* comes in. In Python the following four keywords give you everything you need to do that.

starting a loop

There are two ways to start a loop in Python, and they're conceptually pretty similar, but your choice of which depends a great deal on exactly what you need to do with that loop.

looping until some condition is reached

`while`docsStarts a loop BLOCK by testing a the truth-value of an EXPRESSION; will iterate continuously until EXPRESSION evaluates to False.

From the Old English word ***hwile*** for “a duration of time”, but here we’re using the conjunctive form which implies “[during the] time [that something is true]” or “for as long as [something is true]”. In programming languages the keyword is always associated with something to *test* and something to *do*, so the meaning becomes “***while*** [the test is true], [do something]”.

The form of a `while` loop BLOCK is always the same:

```
1 while EXPRESSION:  
2     STATEMENT
```

If the EXPRESSION evaluates as `True` then STATEMENT will be reached and executed. When the end of the indented BLOCK is reached, control returns immediately to the top, then EXPRESSION is tested again, and so on. So long as EXPRESSION evaluates as `True` the entire indented BLOCK will be run again and again.

The `while` loop can also *optionally* be terminated by an `else` BLOCK:

```
1 while EXPRESSION:  
2     [...]  
3 else:  
4     STATEMENT
```

In this case the STATEMENT inside the `else` block will be executed if the `while` loop runs all the way to completion (its test evaluates to `False`) *without* encountering `break`. This can be useful when there is some cleanup action that needs to occur when a `while` loop has exited naturally.

The `while` is the most basic form of loop, but it's a little dangerous if not used with care. This is because any EXPRESSION that *always* evaluates as `True` will run *forever*. For this reason it's generally important to design the EXPRESSION so that it will *eventually* evaluate to `False`.

looping through the members of a collection

fordocsStarts a loop BLOCK that will iterate *once* over a COLLECTION, visiting every item in it. Can also be marked with `async`, to start an `async for` loop, see below.

An Old English word via the German *für* with a great many meanings; the general meaning is taken from a prepositional sense of “[performing an action] on behalf of [some thing]”. In computing, though, the meaning is actually taken from the contraction of the word *for* with either *every* (meaning “each [item] in a group”) or *each* (meaning “all [of a group]”) to form **for every** or **for each**, both of which mean “[to perform an action] on behalf of each item [in a group]”. In programming languages that descend from ALGOL *for* has traditionally been the most common name for such a loop, with *do* used in a smaller number of languages. Python takes the name from the traditional usage in ALGOL via C, however it is more accurate to describe Python’s version as a **foreach** loop, because there is no explicit counter and the thing being looped over must be *iterable*.

The usage of `for` consistently involves assigning a user-assigned NAME to every OBJECT in a COLLECTION.

```
1 for NAME in COLLECTION:  
2     STATEMENT
```

Thus `for` every item in the COLLECTION the NAME will refer to that item *inside* the scope of the BLOCK; this allows the STATEMENT to use NAME to act on that thing.

This can be a little easier to grasp if you think of `for` as a specialized form of `while` loop:

```
1 while [items remain in COLLECTION to visit]:  
2     NAME = [increment to the next item]  
3     STATEMENT
```

But obviously the `for` loop is much simpler to work with, as you don't need to worry about implementing the machinery necessary to track the start and stop conditions of the loop. You'll find that you can, and usually should, use a `for` loop whenever you're visiting the individual contents of a COLLECTION, rather than risk a runaway `while` loop.

The `for` loop can also *optionally* be terminated by an `else` BLOCK:

```
1 for NAME in COLLECTION:  
2     [...]  
3 else:  
4     STATEMENT
```

In this case the STATEMENT inside the `else` block will be executed if the `for` loop has run to completion *without* encountering `break`. This can be useful when there is some cleanup action that needs to occur when a `for` loop has exited naturally.

controlling a loop

You don't always want to simply run the loop to completion; there may be good reason to exit early, or to skip a round of the loop.
`break`docsUsed to immediately interrupt the current loop iteration, ending the BLOCK it is found within. For this reason must *only* be used within a loop BLOCK.

From the Old English word ***brecan***, which has several forms and meanings. The noun form generally means "to damage, destroy, or render unusable", as in "to ***break*** a leg". Here, however, we use the alternative meaning "to interrupt [a continuous sequence]", as in "to ***break*** an electrical circuit". In programming it specifically means to interrupt a loop from inside that loop.

The `break` statement always forms a line of its own, and it *must* be used in either a `for` or `while` loop. The most common use is to stop a loop immediately if some particular condition is reached:

```
1 while True:  
2     if EXPRESSION:  
3         break  
4     STATEMENT
```

```
1 for NAME in COLLECTION:  
2     if EXPRESSION:  
3         break  
4     STATEMENT
```

This is particularly useful if there's some situation that warrants stopping the loop before it would normally be completed.
`continue` immediately skips the rest of the current loop BLOCK, allowing the loop to continue on to the next iteration. For this reason must *only* be used within a loop BLOCK.

A Middle English verb borrowed via the Old French ***continuer*** from the Latin ***continuare***. The general meaning used here is to “go forward or onward”, “carry on”, or “proceed”. In programming it means to cause a loop to start executing the next iteration, skipping any instructions that follow it.

The `continue` statement always forms a line of its own, and it must be used in either a `for` or `while` loop. The most common use is to skip to the next iteration of a loop immediately when some particular condition is reached:

```
1 while True:  
2     if EXPRESSION:  
3         continue  
4     STATEMENT
```

```
1 for NAME in COLLECTION:  
2     if EXPRESSION:  
3         continue  
4     STATEMENT
```

This is particularly useful if there's some situation that warrants skipping the current loop; for example if you only wanted to act on every second iteration.

Three for Importing Other Things

All of the above, plus the *builtin* functions we'll talk about in a later article, are sufficient to let you start using Python as a *scripting* language, where you glue together things others have written with your own code to do some task you want to accomplish. But you need to be able to access those "things others have written" to do so. That's what Python's *import mechanism* is for.
importdocsUsed to bring the functionality of an external MODULE into your own code.

A verb from the Middle English *importen*, via the Old French from the Latin *importare*. The general meaning is to "bring/carry [goods into this country] from abroad". In computing it means to bring or *import* some functionality *exported* by another program written in the same language into the current program.

The most common usage is very simple, the keyword is followed by the MODULE you want to access:

```
import MODULE
```

You can also import multiple MODULEs separated by commas:

```
import MODULE_A, MODULE_B
```

And for organizational purposes you can put parentheses around them as well:

```
import (MODULE_A, MODULE_B)
```

Any NAME inside the imported MODULE(s) can then be accessed within your own program using the *dot access* pattern in the form MODULE.NAME. So, for instance, if you wanted to get the circumference of a circle:

```
1 import math
2
3 radius = 3
4 circumference = math.tau * radius
```

Usually you'll find all the `import` usage at the top of a MODULE, which makes it pretty easy to determine where such functionality comes from. `from` modifies `import` to allow you to import specific NAMES from within an external MODULE. Can also be used with `raise` and `yield`, see below.

A Middle English word from the Old English *fram*, here we use the preposition form, with a general sense of "departure or movement away [from something]"; in computing we use a more specific sense of "taken **from** a source".

It's used to modify `import` to import a specific NAME from a MODULE, rather than the *entire* MODULE:

```
from MODULE import NAME
```

If you wish to import more than one NAME they can be separated by commas:

```
from MODULE import NAME_A, NAME_B
```

And for organizational purposes you can put parentheses around them as well:

```
from MODULE import (NAME_A, NAME_B)
```

In all cases you can then use the imported NAME directly, so for instance if you want the area of a circle:

```
1 from math import tau
2
3 radius = 3
4 area = tau * radius ** 2 / 2
```

The main reason for `from` is to remove the need to have many references to a MODULE peppered throughout your code, but it's best reserved for when the MODULE has many NAMES that it exports and you want to just use one or two. asModifies `import` to create an alternative NAME (or alias) for an imported NAME. Can also be used with `except` and `with`, see below.

From the Old English **eallswā** meaning “just so” or simply “all so”, which makes it a reduced form of **also**. The usage here comes from the adverb form meaning “[to act] in the manner or role [of some other thing]”. In Python it very specifically means “[from here on refer to this thing] **as** [this instead]”.

Because `from` exists there are two forms of usage:

```
1 import MODULE as MODULE_ALIAS
2 from MODULE import NAME as NAME_ALIAS
```

The point of `as` is to allow you to `import` some MODULE or NAME but refer to it by some other name. This is useful if the original name is particularly long, would conflict with one already in use in your code, or could simply use some extra information in context.

```
from math import pi as half_as_good_as_tau
```

Five for Exceptional Situations

Now that you've got the basics down, you're getting into more complicated territory. What happens if you find yourself reaching a point in the code where you're in an obvious error state and you don't want to continue? This is where the notion of exception handling come into play. We'll go into more detail on the standard Exceptions in later articles, but for now let's just say that an EXCEPTION is a special type of VALUE that signifies a specific issue that has come up

in your program. That issue *might* be an *error*, in which case you might want to crash out of the program, or it might be a signal that something *unusual* but expected has occurred. Either way you need to be able to emit those signals from your own code, as well as catch and react to such signals emitted by other code.

to signal that there's a problem

`raisedocs`Used to raise a specified EXCEPTION, which will cause the program to stop immediately and exit if not handled by an `except` BLOCK. If used without an argument inside an `except` or `finally` BLOCK, re-raises the EXCEPTION being handled by the BLOCK.

A Middle English word with *many* meanings, but in this case it comes from the verb form meaning “to lift upright, build, or construct” or “to make higher”. The meaning in computing is more specifically from a newer sense of “to mention [a question, issue, or argument] for discussion”, as in “to **raise** attention [to an issue]”. In several other programming languages **throw** is used with similar meaning.

The usage is *usually* going to be:

```
raise EXCEPTION
```

Quite rarely you might see the *chained* form:

```
raise EXCEPTION from OTHER_EXCEPTION
```

Which is used to indicate that the EXCEPTION being raised was *caused by* (or *came from*) some other EXCEPTION. This isn't used often, but sometimes is useful when you attempt to handle an exception raised by other code but somehow cannot do so.

Lastly *inside* an `except` BLOCK you can simply:

```
raise
```

Which will re-raise whatever EXCEPTION is currently being handled inside that BLOCK.

to signal that a particular condition is not met

assertdocsUsed to test if some EXPRESSION is considered True, and if not raise an AssertionError.

From the Latin **assertus**, with the general sense of “declared, protected, or claimed”. In programming it specifically means to specify that a condition must be met at a particular point in the code, and to error if it is not.

The usage of `assert` is usually going to be in the form:

```
assert EXPRESSION
```

Which will simply `raise` an AssertionError if EXPRESSION does not evaluate as `True`.

An alternative form allows you to specify a message:

```
assert EXPRESSION, "something is wrong!"
```

And this message will be incorporated into the AssertionError.

The `assert` statement exists for you to test the things that *must* be `True` for your program to continue to work (we call these your *invariants*). This can be very helpful when developing and debugging, but should not be relied on, *at all*, in production code, as the person running your program can elect to disable all `assert` statements by passing the `-O` command line option to Python. Basically you can imagine that `assert` works like:

```
1 if [the -O command line flag was not passed]:  
2     if not EXPRESSION:  
3         raise AssertionError
```

... except that if the `-o` flag was passed the `assert` statement will simply be replaced by `pass` and nothing whatsoever will happen.

to catch a signal and react to it

With both `raise` and `assert` in your toolkit you know how to signal an EXCEPTION, but how do you catch and react to (or ignore) one? This is where Python's exception handling mechanism comes into play. trydocsStarts an exception handler BLOCK; *must* be followed either an `except` BLOCK, an `else` block or a `finally` BLOCK in order to be valid syntax.

A verb borrowed from the Old French word ***trier***, meaning to “test”, “experiment”, or “attempt to do”. The meaning in Python is essentially the same, “[start a] test [of something that *may* error]”. exceptOptionally *continues* an exception handler BLOCK by catching EXCEPTIONs; can (and *should*) be limited to specific types of EXCEPTION. More than one of these can follow a `try` and each will be checked in turn until either the EXCEPTION is handled or no more `except` statements remain.

A verb borrowed from the Middle French ***excepter*** and Latin ***exceptus***; the original meaning is “to receive”, but the more general uses it to “exclude [something]” or “object to [something]”, as in “every fruit except apples”. The meaning in Python is a little vague, but can be thought of as “catch, capture, or trap [an exception]”. In fact in many other languages ***catch*** is used for the same purpose; Python's ***except*** is the exception to that rule. finallyOptionally *cleans up* an exception handler BLOCK to provide a means of *always* performing some action whether or not the EXCEPTION was handled. Must follow any `except` BLOCKS that are present, as well as the optional `else` BLOCK if that is also present. If no `except` BLOCK is present then `finally` must terminate the exception handler.

From the Middle English ***fynally*** meaning “at the end or conclusion” or just “lastly”. It implies the very last thing to do in a sequence, which is the meaning here as well.

Exception handling is the first circumstance you've encountered in which one BLOCK *must* be followed by another, and the rules are somewhat more complicated than anything you've yet seen. For instance there are *two* different *minimal syntactically valid* forms of exception handler:

```
1  try:  
2      BLOCK  
3  finally:
```

This form does not actually handle any EXCEPTION raised within BLOCK, it merely ensures that STATEMENT is run in all circumstances. Any EXCEPTION not handled will continue to “raise up” until it is either handled or crashes your program. This is useful for situations in which you want to do the same action (such as closing a file or database connection) whether or not there has been an error.

The second form *does* handle an EXCEPTION:

```
1 try:  
2     BLOCK  
3 except:  
4     STATEMENT
```

Note: because this form will catch *any* EXCEPTION, including several that are used by Python to perform critical operations, you should, practically speaking, **never** use the above form. Instead use the *minimum safe* form:

```
1 try:  
2     BLOCK  
3 except EXCEPTION:  
4     STATEMENT
```

This ensures that the exception handler *only* handles the EXCEPTION *type* specified. The hierarchy of built-in exceptions in Python is complex and touches on concepts we can't cover in this article, but for now just know that `except Exception` is the *least* specific `except` statement you should ever use in production code.

That minimum safe form, however, isn't particularly useful, because the STATEMENT can't actually know what EXCEPTION it's handling. Most of the time you'll see something like this:

```
1 except EXCEPTION as NAME:  
2     STATEMENT
```

Which uses `as` to provide an alias NAME that can be used to inspect the actual EXCEPTION that was raised. This can be very helpful for seeing precisely what went wrong, which will help you decide if you can ignore the problem or need to respond in some way.

You can also specify multiple different types of EXCEPTION to catch, which can be helpful if you want to respond to any of those forms in the *same* way:

```
1 except (EXCEPTION_A, EXCEPTION_B, EXCEPTION_C) as NAME:  
2     STATEMENT
```

And if you want to respond to them each in a *different* way you can just stack `except` BLOCKS:

```
1 except EXCEPTION_A as NAME:  
2     STATEMENT_A  
3 except EXCEPTION_B as NAME:  
4     STATEMENT_B
```

You can also optionally use `else` with an exception handler, so long as at least one `except` statement is used, and as long as the `else` is positioned *before* any `finally`:

```
1 try:  
2     BLOCK_A  
3 except EXCEPTION as NAME:  
4     BLOCK_B  
5 else:  
6     STATEMENT
```

However this is relatively rare, because the STATEMENT indented under `else` will *only* be executed when no EXCEPTION was raised inside the `try` BLOCK *and* no `break`, `continue`, or `return` statements were encountered within it.

As you've seen exception handling has some fairly dense syntax compared to the rest of Python. A fully fleshed out exception handler in a program that does something fairly complex,

like interacting with a database, might involve all these parts.

```
1 try:
2     [connect to database]
3     [query the database]
4 except ConnectionError as error:
5     [log the error]
6 else:
7     [log success]
8 finally:
9     [close the connection]
```

But, thankfully, it's often not that complex, and you usually only have to deal with exception handling when something truly exceptional has happened.

Four for Writing Functions

Now that you've got all the structures you need to write an arbitrarily complex program, with the ability to make decisions, loop, and handle errors, your biggest problem is going to be organizing those structures into re-usable units. For instance you don't want to type out a complex exception handler for every single time you connect to and query a database, as that would quickly lead to an unmanageable amount of repetition.

One of the key mantras of programming is **Don't Repeat Yourself** (aka **DRY**); the less boilerplate, the better, and so you want to be able to create subroutines, which are the most basic form of *code re-use*. In Python the most common form of subroutine is the *function*, which comes in two primary forms.

anonymous (unnamed) functions

A lot of people teaching (and learning) Python tend to skip over anonymous functions, or treat them as an advanced feature, but that's really just because they're poorly named. The "anonymous" part just means the function isn't given a specific name at creation time. That might not sound like the easiest thing to re-use – and you're right – but it's useful to understand them before we head on to *named* functions because they're fundamentally simpler and more constrained.

Unfortunately the powers that be decided that anonymous functions, already saddled with a bad and confusing name, should *definitely* get a worse name in Python.`lambda`docsUsed to

define a CALLABLE anonymous function and its *signature*.

The 11th letter of the Classical Greek alphabet, λ ; it has no general meaning in English. In Python it is used because anonymous functions are a fundamental unit of Alonzo Church's Lambda Calculus, which provides much of the mathematical underpinnings of modern computation. As an honor, that's nice; in reality *lambda* would be used more often if it had a more fun name.

Despite the name a `lambda` is actually the conceptually simplest form of function in Python, because you can think of it as just a way of creating a delayed-evaluation EXPRESSION that's stated with a specific form:

```
lambda : EXPRESSION
```

Which evaluates to a CALLABLE that will evaluate the EXPRESSION that comes *after* the colon only when the CALLABLE is itself called. To call it you use the *call syntax* that is common to all Python CALLABLEs, however since the `lambda` is itself an EXPRESSION you need to surround it with parentheses to do that.

```
(lambda : EXPRESSION)()
```

And *voila*, everything you've just written will instantly be replaced by whatever the inner EXPRESSION evaluates to.

Thus all three of these are entirely identical:

```
1 x = (lambda : 2 + 2)()
2 x = 2 + 2
3 x = 4
```

But just delaying an EXPRESSION isn't really the most useful tool. So the `lambda` introduces the idea of a *function signature*: you can add a NAME to the left side of the colon and the thing

that NAME refers to will able to be used inside the inner EXPRESSION when it evaluates. This name is known as a *parameter* of the function.

Here's a `lambda` with a single parameter:

```
lambda NAME: EXPRESSION
```

And here's one with two parameters:

```
lambda NAME_A, NAME_B: EXPRESSION
```

And when we want to use the `lambda` we just call it, passing in a concrete VALUE for every parameter in the signature:

```
(lambda NAME_A, NAME_B: EXPRESSION)(VALUE_A, VALUE_B)
```

Which you can imagine can be useful if we want to calculate the area of a rectangle:

```
lambda width, height: width * height
```

But, again, we're going to end up writing that a lot if we don't assign it to a NAME:

```
1 area = lambda width, height: width * height
2 square = area(2, 2)
3 rectangle = area(3, 5)
```

Great! The `square` is now `4` and `rectangle` is `15`; we've got the basis of code re-use!

But `lambda` is going to get pretty nasty when the EXPRESSION starts to get long. And beyond that there are some pretty significant limitations, since we actually *cannot* execute many kinds of STATEMENT within a `lambda`, much less an arbitrary BLOCK. They have their place, but maybe there's a better and more flexible way?

Since we've just gone and assigned what is supposed to be an *anonymous* (which means "has no name") function to a NAME, let's look at how we should *usually* write functions in Python.

named functions

The *named* function builds on the ideas of the `lambda` but take them to a *much* more flexible place, allowing you to re-use large chunks of code quite easily. They're a little more subtle to master too, because they don't evaluate exactly like an EXPRESSION. In fact by default they'll always evaluate to `None` unless you explicitly tell them to do otherwise. defdocsUsed to define a *named* function and its *signature*, the indented BLOCK that follows can then be re-used by calling that NAME using the `function()` syntax. If used inside a `class` defines a named *method* instead, which is called using the `class.method()` syntax. Can also be marked with `async`, to start an `async def`, see below.

A contraction of the word **define**, which comes via the Middle English **deffinen** from Old French and Latin roots. It's a verb that means "to specify or fix [the meaning of a word or phrase]". In Python it is used specifically to create a named subroutine. In other languages **define**, **fn**, **fun**, **func**, **function**, and **let** are often used instead.

Because `def` can be used to create arbitrarily complex CALLABLEs, it's going to require some explanation. Essentially it's used to create a new NAME that refers to a CALLABLE. In fact, from now on let's just use FUNCTION to mean exactly that:

```
1 def FUNCTION():
2     STATEMENT
```

This defines a FUNCTION that can be called using `FUNCTION()`, which will then execute every STATEMENT inside the indented BLOCK. As you can see it's conceptually equivalent to the "named" `lambda` form you saw earlier. Compare these two forms:

```
1 def FUNCTION(NAME_A, NAME_B): STATEMENT
2 FUNCTION = lambda NAME_A, NAME_B : STATEMENT
```

And you'll see they're *very* similar, except that the `def` version can run an arbitrary number of STATEMENTS³.

These STATEMENTS will be executed within the *local scope* of the function, meaning that *any* NAME assigned *inside* the function cannot be *directly* seen or accessed by any code *outside* the function. In this example there are no such names, but there will be as soon as we change the *signature* of the function:

```
1 def FUNCTION(NAME):
2     STATEMENT
```

And just as with `lambda` we can have multiple parameters:

```
1 def FUNCTION(NAME_A, NAME_B):
2     STATEMENT
```

Which let's us build the `area` function we made before with a `lambda` :

```
1 def area(width, height):
2     width * height
3
4 square = area(2, 2)
5 rectangle = area(3, 5)
```

Except, wait a minute ... didn't I say just above that `def` functions evaluate to `None` when called? Yep, right now both `square` and `rectangle` are assigned to `None`. We've calculated two areas, but then discard them again as soon as the *local scope* of `area` is closed. Now, how the heck do we get the area *out of* `area`, anyway?

stop the function and give back a value

`return`docsUsed to immediately give up control and end execution of the function at the point at which it is encountered. If followed by an EXPRESSION, that is evaluated first and the resulting OBJECT is given back to the caller of the function. If no EXPRESSION is present `None` is returned instead. Has no meaning outside a function, thus if present at all it *must* be inside a BLOCK that follows a `def`.

A Middle English verb from the Old French *retourner*, meaning “to turn back” or “to go back [to a former position]”. In computing it has two meanings:

1. The intransitive meaning is “to give back (or relinquish) control [to the calling procedure]”, as in “when the function exits it will *return* control”.
2. The transitive meaning is “to pass [some data] back to the calling procedure”, as in “this function will *return* the current time”.

In Python both meanings are combined, since a function will always *return* both control *and* data to the caller.

The most basic usage of `return` immediately ends the function and returns `None` to the caller:

```
return
```

This form is relatively rarely used; as mentioned before a function *always* evaluates to `None` by default; thus you can imagine that `return` is the implicit last line of *every* function.

Because of that it is much more common to see:

```
return EXPRESSION
```

Which immediately evaluates the EXPRESSION and returns the resulting OBJECT back to the caller.

You can also use `return` to pass back more than one OBJECT:

```
return EXPRESSION_A, EXPRESSION_B
```

This evaluates each EXPRESSION in turn, ends the function, and returns a **tuple** (a type of fixed-length COLLECTION) containing one OBJECT for each EXPRESSION.

This multiple form is also relatively rare, as it can be a bit of a surprise to the user of the code, and so requires a bit more effort in documentation, but it can be convenient for internal functions that you don't intend to be used by others.

What you'll notice is that `return` *terminates* a function at the moment it's evaluated. Anything below that point in the function essentially doesn't exist (with one exception, the `finally` block inside an exception handler). So how would you handle situations in which you needed to give back data, but continue to work afterwards?

pause the function and give back a value

`yield`docsUsed to immediately pause execution and temporarily give up control at the point at which it is encountered. If followed by an EXPRESSION, that is evaluated first and the resulting OBJECT is yielded back to the caller of the function; if no EXPRESSION is present `None` is yielded instead. Has no meaning outside a function, thus if present at all it *must* be used inside a BLOCK that follows a `def`. Can be modified by `from` to form `yield from`, see below.

Etymologically the oddest word in this list; derives from the Middle English ***yielden*** and the Old English ***gieldan***, both of which mean “to pay”, and share their root with the Old Norse ***gjald*** and the German ***geld***, both of which mean “money”. Today ***geld*** means an ancient form of compelled tax or ransom, but it also means “to castrate”. Historically the Dangeld was a tax raised on the English by their king. This tax was raised to pay waves of Danish Vikings to, presumably, not castrate the English (or, at the very least, their king). None of this is directly important here, but might explain a little of why the meaning in computing derives from both “to give way and relinquish control”, as in “***yield*** to oncoming traffic”, and “to give back [a result or return on investment]” as in “the fund has a ***yield*** of 5% per year”. In both cases the implication is that the situation is not yet final, and is likely recurring: you ***yield***, then you wait, then you ***yield*** again.

In Python it helps to remember an allegory: `return` is Death, and comes exactly once, while `yield` is Taxes, and may only end when Death arrives.

The usage of `yield` is similar to `return` :

```
yield
```

```
yield EXPRESSION
```

```
yield EXPRESSION_A, EXPRESSION_B
```

However *any* function that uses `yield` actually returns a special kind of COLLECTION known as a GENERATOR when called. Unlike a normal COLLECTION a GENERATOR does not hold all of its items in memory simultaneously, but instead “generates” each item as required by running through the function until `yield` is encountered and it gives forth an item. The GENERATOR pauses execution at that point, allowing the code using the GENERATOR to work with that item. When desired it can request the next item, at which point the GENERATOR continues running to the next `yield`, and so on until there are no more `yield` statements left to run or a `return` is encountered.

This is what allows the next form:

```
yield from GENERATOR
```

Which is a specialized use case that allows you to write a GENERATOR that will `yield` every item, in turn, from *another* GENERATOR.

A thorough explanation of GENERATORS is outside the scope of this series of articles, as they’re a fairly advanced topic. For now it’s sufficient to know that if you encounter `yield` you’re looking at a GENERATOR.

Three for Manipulating Namespaces

Now that you understand the basics of function, you'll need to know about namespaces, as they become very important when you start building more complex programs.

At its simplest a namespace is essentially just the place where Python stores any NAME you create, so they're used whenever Python needs to look up the OBJECT referred to by that NAME. Each namespace has a *scope*, which limits how long the namespace "lives" and whether or not any NAME within it is *readable* and/or *writable* by a given STATEMENT. Namespaces form a hierarchy, however, so the rules can be a bit tricky to remember.

In any given namespace a STATEMENT can:

1. both *read* and *write* any NAME defined in its own namespace
2. *read*, but not *write*, any NAME defined in an enclosing *parent* namespace
3. neither *read* nor *write* any NAME defined in a *sibling* or *child* namespace

At the top level is Python's own namespace, which contains all the *builtin* NAMEs that Python provides for you. This namespace *cannot* be written to, and any NAME in it essentially lives forever.

Next comes the *global* namespace, which is the namespace of the MODULE you're working in; it becomes the parent of all other namespaces created within that MODULE. Any NAME defined here *usually* lives for the duration your program is running.

Next, each CALLABLE you create gets its own unique *local* namespace *when it is called*; any NAME created here lives only as long as the CALLABLE is running, and when it finishes both the NAME and the OBJECT it refers to will be destroyed, unless either `return` or `yield` is used to pass it back to the calling scope.

Since you can define a CALLABLE that is nested *inside* another CALLABLE you can build a namespace hierarchy of arbitrary depth, so any given NAME might exist in any number of namespaces further and further removed from where you're actually trying to use that NAME. This can get unwieldy to think about pretty quick, though, which is why we *try* to limit the amount of nesting we do in practical code.

The assignment of a new NAME is pretty straightforward: unless modified by one of the keywords below a NAME is *always* assigned in the scope in which the assignment STATEMENT occurs.

But the *use* of a NAME is a little less obvious: when used in a STATEMENT Python first searches for the NAME in the immediate *local* scope. If it cannot find the NAME then it searches the immediate parent (known as the *nonlocal*) scope, and then it keeps searching each successive parent scope until it either finds the NAME or cannot resolve it and errors.

All of this usually somewhat invisible machinery exists to allow you to write code using NAMES that are appropriate and clear to the scope in which you're working. So long as it isn't a keyword and you don't *need* to use the NAME from an ancestor's scope you can use whatever NAME you want locally and not worry about it overwriting anything *outside* its own scope.

Most of the time you don't want to fool around with that machinery, but every so often there's a good reason to. Or a bad one that's gotten you confused.

write to the top from anywhere

In *extremely rare* circumstances you want to directly manipulate the *global* namespace in a way you wouldn't normally be allowed. Used to declare a NAME as part of the *global* namespace; the NAME *cannot* have been used previously in the same namespace. In effect this allows a *local*/STATEMENT to both create and assign to a *global*/NAME it otherwise could only read. *Can* be used within the global namespace, but has no effect.

An adjective borrowed from the French, with the general meaning of "worldwide" and "universal". In computing the meaning is specifically "[a NAME that is] accessible by all parts of a program". In Python it is both the top-most *namespace*, which is *readable* from within all functions and methods, but is also a keyword that binds a *local*/NAME into the *global* namespace, allowing it to be *writable* from within that *local* namespace.

The usage is quite simple; the main restriction is that it cannot appear *after* the NAME has been used:

```
1 global NAME
2 NAME = EXPRESSION
```

You should be aware that the use of `global` is an *immediate* code smell, as its use is *almost always* an indication of bad coding habits, because it *usually* can – and almost certainly *should* – be replaced with *argument passing* and *return values*. There are data structures and

dynamic code generation tasks that would be impossible to build without it, but the *odds* are if you're thinking about using it, **don't**.

write to the parent from the child

Slightly more often there's a need for the child to tell the parent what to do.`nonlocal`docsUsed exclusively inside *nested* functions / methods (also known as *closures*) to bind a *pre-existing* NAME in the *parent's* namespace. This allows a STATEMENT in the child to have write access to a NAME defined in its parent. *Must* appear before any reference to that NAME in the local scope.

An adjective formed by combining the prefix ***non-*** ("not") and ***local*** ("pertaining to a particular place"). This is a primarily scientific and technical *neologism* (literally "new word") that has no true general meaning, only specific meanings within specific fields. In programming it signifies a non-local variable, meaning "[a NAME that is] accessible in neither the *local* nor the *global* namespace". In Python it very specifically applies to the namespace of the enclosing function from the point of view of a nested function, and is also a keyword that binds a *local* NAME within that nested function into the ***nonlocal*** namespace, allowing it to be *writable* from within the nested function.

Because it is *only* of use inside a nested function, usage will always involve some enclosing structure:

```
1 def OUTER_FUNCTION():
2     NAME = VALUE
3     def INNER_FUNCTION():
4         nonlocal NAME
5         NAME = EXPRESSION
6     INNER_FUNCTION()
```

The key thing to understand is that `nonlocal` is used in a *similar* fashion to `global`, but with the more limited goal of making a variable in the parent of a nested function writable. It has similar caveats as well, though the fact that its impact isn't felt by the *entire* program makes it inherently less dangerous. Still, its uses are *very* narrow and *relatively* few, so try not to abuse it when argument passing will do.

kill it with fire

So far you've seen how to *add* a NAME to a namespace, but how do you *remove* a NAME from a namespace? `del` docs Used to delete a NAME (or NAMEs) from a namespace; if no other references exist to the OBJECT that NAME referred to, the underlying OBJECT is deleted as well. Can also be used to delete an attribute of an OBJECT or a member (or members) of a COLLECTION *if* the specific type has allowed this operation.

A contraction of **delete**, which is in turn a verb derived from the Latin **deletus**. Its general meaning is to "destroy or eradicate", "erase or smudge out", and "utterly remove", and while the specific meaning in computing is a little less violent it has similar effect. In Python the meaning is a little indirect: `del` is used to delete a *reference* that the *user* directly controls, which *may* indirectly trigger deletion of the thing referred to from process memory, which the *interpreter* controls.

The most common usage is to provide a single NAME:

```
del NAME
```

However it is also valid to provide multiple NAMES, comma separated:

```
del NAME_A, NAME_B, NAME_C
```

In either case Python will proceed left to right, deleting each NAME from the namespace in which the `del` invocation has occurred.

Because the effect is limited to the *closest* namespace, you have to use `global` or `nonlocal` to delete a NAME outside the local namespace:

```
1 global NAME
2 del NAME
```

You can also use `del` to delete attributes of OBJECTs and member(s) of COLLECTIONs, however what actually happens depends on the type of the OBJECT / COLLECTION involved.

The **list** COLLECTION, for instance, allows you to use `del` to delete both individual *indices* and *slices* of its contents:

```
1 test = [1, 2, 3]
2 del test[0]           # deletes the first item
3 del test[:]          # deletes all items
```

We'll leave a further exploration of this for when we discuss the *builtin* types in later articles. In the meantime, let's finally meet the keyword that will let you start defining such type-specific behaviors for yourself.

One for Defining New Types of Object

Up until this point I've been pretty vague about what an OBJECT actually is. In the Object-Oriented Programming paradigm, also known as OOP, an object is, basically, a *thing* that has both *state* (in the form of named *attributes*) and *behavior* (in the form of callable *methods*). Two individual things of the same *type* may have different specific values for those attributes – we call them different *instances* of the same *type* – but they share the same overall *interface*. However there's a bit more to it than that; the programmer should be able to define partial interfaces, which we'll call *traits*, that types with similar needs can implement in different ways, such that they all share some common attributes and behaviors (a property known as *polymorphism*). Additionally these traits should be able to be passed from more generic types to more specific types via an inheritance mechanism, much as a parent passes on traits to their children.

You can take an example from nature; a duck is a type of bird, and it inherits certain traits it shares with all birds. All birds are a type of animal, and thus inherit certain traits shared with all animals. An animal is a type of life, and so on. Thus Howard, a specific duck, has all the attributes and behaviors of life, animals, birds, and ducks, all rolled up nicely in one instance of the duck type.

In fact it is from just such an example of biological *classification* that Python takes the keyword it uses for defining new types of OBJECT.`classdocsUsed` to define a template for a new type of OBJECT.

A noun borrowed via the French **classe** from the Latin **classis**, which meant "a division, army, fleet", "the people of Rome under arms", and, oddly specifically, "any one of the six orders into

which Servius Tullius divided the Roman people for the purpose of taxation”, which goes a long way towards explaining why English is such a difficult language to master. The general meaning we use here, though, is loosely borrowed from evolutionary taxonomy, implying “a group that shares certain inheritable traits”. The specific meaning in Python comes from Object-Oriented Programming and implies “a template for creating objects that share common inheritable state (attributes) and behavior (methods)”.

In Python you define a new type by creating a new NAME that is also a CALLABLE; from now on we’ll call that a CLASS. The `class` keyword is used to start a new BLOCK that defines the *implementation* of that CLASS:

```
1 class CLASS:  
2     pass
```

The CLASS can now be called to create a new instance of its type:

```
NAME = CLASS()
```

Now from the above you might expect that instance to have *no* attributes or behavior, but in fact it does have the *minimal* interface that all OBJECTs share (it can be printed and used in comparisons, for instance, though not very usefully). It receives this because *all* new `class` definitions implicitly inherit from **object**, which is Python’s *base type*.

You can also specify a more generic CLASS (also known as a *superclass*) to inherit from:

```
1 class CLASS(SUPERCLASS):  
2     pass
```

In fact you’ll often want to inherit from more than one superclass in order to mix together their various traits. This is done by separating the superclasses with commas:

```
1 class CLASS(SUPERCLASS_A, SUPERCLASS_B):
```

```
2     pass
```

The notion of multiple inheritance can get quite complicated when more than one superclass defines the same attribute or method NAMEs, so for now let's just keep our classes simple and assume no such overlap.

Just defining the CLASS and its superclasses without giving it some attributes and methods of its own is unusual, so you'll *normally* see at minimum a custom *initializer* method, which by convention is done by defining the special *double underscore* (or "dunder") *instance method* `__init__`:

```
1 class CLASS:  
2     def __init__(self, NAME):  
3         self.NAME = NAME
```

As you can see an instance method is just another kind of function definition made using `def`. In fact a METHOD is just a FUNCTION that's been added to the CLASS's namespace. That's right, the `class` keyword also creates a new kind of local namespace, this one remains attached to the CLASS and allows names to be looked up on either the *instance* or the class itself using the *dot access* patterns INSTANCE.NAME and CLASS.NAME. Notice the `self` parameter name in the method definition above? That's the name given by convention to the *first* parameter of any instance method, and it's how you access attributes set on the instance itself.

This all sounds a little abstract, so let's demonstrate it by building a CLASS for thinking about circles.

```
1 from math import tau  
2  
3 class Circle:  
4     def __init__(self, radius):  
5         self.radius = radius  
6  
7     def diameter(self):  
8         return self.radius * 2  
9  
10    def circumference(self):  
11        return self.radius * tau
```

```
12
13     def area(self):
14         return tau * self.radius ** 2 / 2
```

Now if we create an instance representing the *unit circle*:

```
unit = Circle(1)
```

We can access and **print** the **self.radius** attribute:

```
print(unit.radius)
```

And we can also call any of its methods, but notice how we never have to pass the radius of the circle as an argument, since the methods can all access that via **self**:

```
1 print(unit.diameter())
2
3 print(unit.circumference())
4 print(unit.area())
```

Since they're the fundamental building block of *everything* in Python's data model, there's a *lot* than can be said about classes in Python, and there's a lot of subtleties that can go into their proper use (and improper abuse), so we'll come back to them often in later articles in this series. For now though the important thing is to start to recognize how `class` is used to create them, and how instances are used when accessing their attributes and methods. It's also helpful to know that there are quite a few "dunder" methods, and that these are used to implement a lot of the common "under-the-hood" functionality that supports all the builtin functions we'll start to meet in later articles.

One for Working Within a Context

Sometimes there are actions you *always* want to perform within a specific context, such as committing a database transaction or closing a network connection. These sort of things *usually* involve some form of *exception handling* or other boilerplate specific to the task, and that can lead to a lot of boilerplate that needs to be repeated. An example is something as simple as reading a file: opening the file must necessarily mean getting some resources from the underlying operating system, and you always want to free up those resources, even if you accidentally tried to open a file that didn't exist or which you did not have permissions to read. The file is a *type* of OBJECT that will always carry that contextual need to manage a resource with it, and so you'll need to write the same `try` and `finally` boilerplate *every time* you open a file.

Unless the language you're working in provides a convenient means of ensuring that it happens for you.`with` starts a *context manager* BLOCK, which ensures that the indented STATEMENT(s) below it are performed within the context of the OBJECT being managed. Can also be marked with `async`, to start an `async with`, see below.

A Middle English preposition that takes its pronunciation from one Old English term, *wið* ("against"), but takes its modern meaning from another, *mid* ("in association with") which in turn comes from the German *mit*. The meaning in Python derives from *within*, meaning "inside the scope or context of [some thing or event]".

Any use of `with` requires an EXPRESSION that evaluates to an OBJECT that satisfies the context manager type's interface, and thus has implemented both the `_enter_` and `_exit_` "dunder" methods. From now on we'll call that EXPRESSION a CONTEXT_MANAGER, to make the examples clearer.

In the most basic usage you simply start a new BLOCK:

```
1 with CONTEXT_MANAGER:  
2     STATEMENT
```

Which will enter the context by calling `CONTEXT_MANAGER.__enter__()`, execute the STATEMENT within the context, and then exit the context by calling `CONTEXT_MANAGER.__exit__()`.

But most often you'll want to use `as` to assign the OBJECT returned by `CONTEXT_MANAGER.__enter__()` to an alias NAME so the BLOCK can work with it:

```
1 with CONTEXT_MANAGER as NAME:  
2     STATEMENT
```

This can save you a lot of boilerplate, especially with the many standard OBJECTs that already implement the context manager interface. For instance the recommended way to read a file's contents:

```
1 with open(path) as src:  
2     contents = src.read()
```

Is *much* simpler than what you'd otherwise have to write:

```
1 src = open(path)  
2 try:  
3     contents = src.read()  
4 finally:  
5     src.close()
```

Occasionally you'll want to work within more than one CONTEXT_MANAGER:

```
1 with CONTEXT_MANAGER_A as NAME_A, CONTEXT_MANAGER_B as NAME_B:  
2     STATEMENT
```

Which works exactly the same as nesting one `with` inside another:

```
1 with CONTEXT_MANAGER_A as NAME_A:  
2     with CONTEXT_MANAGER_B as NAME_B:  
3         STATEMENT
```

You'll get to meet quite a few context managers as you work your way through the Python standard library. And now you know that building your own is just a matter of creating a new

`class` and implementing a couple of “dunder” methods.

And, Finally... Two for Working Asynchronously

Any function or method you write with `def` alone is, by definition, *synchronous*, meaning that the moment you call it your running code has to stop everything else and *wait*, however long it takes, for your function to either `return` or `yield` control back to it before the rest of your code can be executed. For most tasks that happen on a local machine this is sensible and perfectly fine, especially when you *need* the answer before you can proceed any further.

In much modern programming though it's often necessary to interact with things that *don't* respond quickly, like network interactions with servers that could be a world away, and sometimes there's *plenty* of work you could be doing while you wait for them to respond. You still *need* the answer, but you need it *eventually*. One way this can be addressed is via asynchronous programming, which as of Python 3.5 has become a first-class part of the Python language.

Asynchronous programming is an advanced topic that is fairly specialized, so using it is well outside the scope of this series. It's also quite a new addition to the language, and its usage has fluctuated from version to version. Rather than try to summarize those changes I'll just describe the very basics of using these keywords in Python 3.7 below. If, however, you're curious there's a quite good `async` primer available here that seems to be kept up to date.

Because asynchronous code does the same work as synchronous code, but *schedules* the execution of that work differently, only two new keywords needed to be added to the language.`async` used to mark another KEYWORD as one that works asynchronously. As such, `async` *cannot* appear on its own. With `def` as `async def` to define an asynchronous function, also known as a COROUTINE. With `for` as `async for` to loop over an *asynchronous iterator* inside an `async def`. With `with` as `async with` to use an *asynchronous context manager* inside an `async def`, see below.

As you may have guessed, this is a contraction of the Modern English word ***asynchronous***, which is formed by combining the Latin roots ***a-*** (“not”) and ***syn-*** (“together”) with ***Khronus***, the Ancient Greek personification of *time*. It unambiguously means “not occurring at the same time”. In Python it more specifically marks an operation as “not occurring in the same time as the caller”, which allows the caller to wait for the result of that operation, which will occur at *some* point in the future.

Because the meaning of `async` is tied to the KEYWORD it marks, its usage is always the same:

```
async KEYWORD
```

However, unlike other any other keyword we've seen, the usage of `async` will *always* begin with the definition of a new COROUTINE:

```
1 async def COROUTINE():
2     STATEMENT
```

Both of the other forms of `async` exist to allow you to work *within* a COROUTINE with *other* COROUTINES, and thus they can only exist *inside* an `async def`.

For instance you can use `async for` to loop over an asynchronous iterator such as a GENERATOR COROUTINE (which is simply a COROUTINE that uses `yield` instead of `return`).

```
1 async def COROUTINE():
2     async for item in GENERATOR_COROUTINE:
3         STATEMENT
```

You can also use `async with` to perform work within the context of an asynchronous context manager:

```
1 async def COROUTINE():
2     async with CONTEXT_MANAGER_COROUTINE as NAME:
3         STATEMENT
```

Virtually every COROUTINE will need to wait on other COROUTINE(s), which is why there's another keyword that can only be used within an `async def`. `await` is used to suspend the

execution of the COROUTINE it is found *within* and waits for the COROUTINE to its right to complete; can only be used inside an `async def`.

From the Middle English verb ***awaiten*** (“to wait for”) from the Old French ***awaitier/agaitier*** (“to lie in wait for, watch, or observe”). The general sense is more active and hostile than ***wait***, which it’s a modification of. In asynchronous programming it means to “[suspend execution] and wait [for something to finish]”.

The `await` keyword is always going to be used to call a COROUTINE inside an `async def`:

```
1 async def OUTER_COROUTINE():
2     await COROUTINE()
```

Which will pause the execution of the outer COROUTINE and wait until the called COROUTINE, eventually, returns control. Any VALUE returned by the called COROUTINE is passed through `await`, so you can think of `await COROUTINE()` as an asynchronous EXPRESSION that will, *eventually* evaluate to whatever the COROUTINE returns when called.

Of course now that you’ve got the basics of doing asynchronous work down, how do you actually *perform* such work? Well, as of Python 3.7 that still requires using functionality provided by the `asyncio` module, the details of which are *well* outside the scope of these articles. See the `async` primer I mentioned earlier for details.

Whew...

There you have them, Python’s 35 keywords: in and of themselves not enough to make you fluent in Python, but if you’ve read this far and digested it you’re well on your way to truly *understanding* (and not merely *using*) the skeleton of *what is going on* in one of the fastest growing general programming languages around. In the next post in this series we’ll take a step away from *words* and look instead at *symbols*, with a dive into Python’s slightly smaller, but thankfully *much* simpler, list of *operators*.

1. Technically **def** is a final clipping or apocope, a specific kind of contraction. English is hard enough already, so I’ll use the more general term. ↑

2. There are 35 keywords as of Python 3.7, the current major version of the language as of the time of writing. New Python keywords are added quite rarely, and it's even more rare for keywords to be removed, but in whatever version you're on you can use

```
from keyword import kwlist; print(kwlist)
```

to view the current list. ↑

3. Putting `def` on a single line makes the equivalence with `lambda` more obvious, but for the sake of readability don't do this very often. ↑

Intro 2 Python3

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
1 # this is the first comment
2 spam = 1 # and this is the second comment
3         # ... and now a third!
4 text = "# This is not a comment because it's inside quotes."
```

Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses `()` can be used for grouping. For example:

2 + 2 4 50 - 5_6 20 (50 - 5_6) / 4 5.0 8 / 5 # division always returns a floating point number 1.6

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use % :

```
| | | 17 / 3 # classic division returns a float 5.666666666666667
```

```
| | | 17 // 3 # floor division discards the fractional part 5 17 % 3 # the % operator returns the remainder of the division 2 5 3 + 2 # floored quotient divisor + remainder 17
```

With Python, it is possible to use the ** operator to calculate powers :

```
| | | 5 ** 2 # 5 squared 25 2 ** 7 # 2 to the power of 7 128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
| | | width = 20 height = 5 9 width height 900
```

If a variable is not "defined" (assigned a value), trying to use it will give you an error:

```
| | | n # try to access an undefined variable Traceback (most recent call last): File "", line 1, in NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
| | | 4 * 3.75 - 1 14.0
```

In interactive mode, the last printed expression is assigned to the variable _ . This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
| | | tax = 12.5 / 100 price = 100.50 price * tax 12.5625 price + 113.0625 round(2) 113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to int and float, Python supports other types of numbers, such as `~decimal.Decimal` and `~fractions.Fraction`. Python also has built-in support for complex numbers _, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result . _ can be used to escape quotes:

```
'spam eggs' # single quotes 'spam eggs' 'doesn\'t' # use \' to escape the single quote...
"doesn't" "doesn't" # ...or use double quotes instead "doesn't" "Yes," they said.' "Yes," they
said.' "\"Yes,\\" they said." "Yes," they said.' "Isn\'t," they said.' "Isn\'t," they said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The print function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
"Isn\'t," they said.' "Isn\'t," they said.' print("Isn\'t," they said.) "Isn't," they said. s = 'First
line.\nSecond line.' # \n means newline s # without print(), \n is included in the output
'First line.\nSecond line.' print(s) # with print(), \n produces a new line First line. Second
line.
```

If you don't want characters prefaced by _ to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
print('C:\some\name') # here \n means newline C:\some name print(r'C:\some\name') #
note the r before the quote C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''` . End of lines are automatically included in the string, but it's possible to prevent this by adding a _ at the end of the line. The following example:

```
1 print("""\
2 Usage: thingy [OPTIONS]
```

```
3      -h          Display this usage message
4      -H hostname  Hostname to connect to
5      """")
```

produces the following output (note that the initial newline is not included):

```
``` {.sourceCode .text} Usage: thingy [OPTIONS] -h Display this usage message -H hostname
Hostname to connect to
```

...

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

## 3 times 'un', followed by 'ium'

```
3 * 'un' + 'ium' 'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated. :

```
'Py' 'thon' 'Python'
```

This feature is particularly useful when you want to break long strings:

```
text = ('Put several strings within parentheses ' ... 'to have them joined together.') text 'Put
several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
prefix = 'Py' prefix 'thon' # can't concatenate a variable and a string literal File "", line 1
prefix 'thon' ^ SyntaxError: invalid syntax ('un' 3) 'ium' File "", line 1 ('un'3) 'ium' ^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
prefix + 'thon' 'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
| | | word = 'Python' word[0] # character in position 0 'P' word[5] # character in position 5 'n'
```

Indices may also be negative numbers, to start counting from the right:

```
| | | word[-1] # last character 'n' word[-2] # second-last character 'o' word[-6] 'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

```
| | | word[0:2] # characters from position 0 (included) to 2 (excluded) 'Py' word[2:5] # characters from position 2 (included) to 5 (excluded) 'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced. :

```
| | | word[:2] # character from the beginning to position 2 (excluded) 'Py' word[4:] # characters from position 4 (included) to the end 'on' word[-2:] # characters from the second-last (included) to the end 'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
| | | word[:2] + word[2:] 'Python' word[:4] + word[4:] 'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of  $n$  characters has index  $n$ , for example:

1	-----+-----+-----+-----+
2	P   y   t   h   o   n
3	-----+-----+-----+-----+
4	0    1    2    3    4    5    6

-6 -5 -4 -3 -2 -1

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from  $i$  to  $j$  consists of all characters between the edges labeled  $i$  and  $j$ , respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
| | | word[42] # the word only has 6 characters
Traceback (most recent call last): File "", line 1,
in IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
| | | word[4:42] 'on' word[42:] "
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
| | | word[0] = 'J'
Traceback (most recent call last): File "", line 1, in TypeError: 'str' object does
not support item assignment
word[2:] = 'py'
Traceback (most recent call last): File "", line 1, in TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
| | | 'J' + word[1:] 'Jython' word[:2] + 'py' 'Pypy'
```

The built-in function `len` returns the length of a string:

```
| | | s = 'supercalifragilisticexpialidocious' len(s) 34
```

## Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type. :

```
| | | squares = [1, 4, 9, 16, 25]
squares [1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
squares[0] # indexing returns the item 1 squares[-1] 25 squares[-3:] # slicing returns a new list [9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
squares[:] [1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
squares + [36, 49, 64, 81, 100] [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
cubes = [1, 8, 27, 65, 125] # something's wrong here 4 ** 3 # the cube of 4 is 64, not 65!
64 cubes[3] = 64 # replace the wrong value cubes [1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append` method (we will see more about methods later):

```
cubes.append(216) # add the cube of 6 cubes.append(7 ** 3) # and the cube of 7 cubes
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] letters ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

---

## replace some values

```
letters[2:5] = ['C', 'D', 'E'] letters ['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

---

## now remove them

```
letters[2:5] = [] letters ['a', 'b', 'f', 'g']
```

---

## clear the list by replacing all the elements with an empty list

```
letters[:] = [] letters []
```

The built-in function `len` also applies to lists:

```
letters = ['a', 'b', 'c', 'd'] len(letters) 4
```

It is possible to nest lists (create lists containing other lists), for example:

```
a = ['a', 'b', 'c'] n = [1, 2, 3] x = [a, n] x [['a', 'b', 'c'], [1, 2, 3]] x[0] ['a', 'b', 'c'] x[0][1] 'b'
```

## First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

### Fibonacci series:

```
1 ... # the sum of two elements defines the next
2 ... a, b = 0, 1
```

```
while a < 10: ... print(a) ... a, b = b, a+b ... 0 1 1 2 3 5 8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the

right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The while loop executes as long as the condition (here: `a < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The print function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
i = 256*256 print('The value of i is', i) The value of i is 65536
```

The keyword argument `end` can be used to avoid the newline after the output, or end the output with a different string:

```
a, b = 0, 1 while a < 1000: ... print(a, end=',') ... a, b = b, a+b ...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

## Footnotes

<sup>1</sup> interpreted as `-(3**2)` and thus result in `-9`. To avoid this and <sup>2</sup> get `9`, you can use `'(-3)**2'`.

- 1 same meaning with both single (`'...') and double ('"..."') quotes.
- 2 The only difference between the two is that within single quotes you
- 3 don't need to escape `"` (but you have to escape `\'`) and vice
- 4 versa.

# Untitled

# STDLib

## Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
import os
os.getcwd() # Return the current working directory 'C:\Python39'
os.chdir('/server/accesslogs') # Change current working directory
os.system('mkdir today') # Run the command mkdir in the system shell 0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open` from shadowing the built-in `open` function which operates much differently.

The built-in `dir` and `help` functions are useful as interactive aids for working with large modules like `os`:

```
import os
dir(os) # help(os) <returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
import shutil
shutil.copyfile('data.db', 'archive.db') # archive.db
shutil.move('/build/executables', 'installdir') # installdir
```

## File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
import glob
glob.glob('*.*py') # ['primes.py', 'random.py', 'quote.py']
```

## Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance the following output results from running `python demo.py one two three` at the command line:

```
import sys
print(sys.argv) # ['demo.py', 'one', 'two', 'three']
```

The argparse module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
1 import argparse
2
3 parser = argparse.ArgumentParser(prog = 'top',
4 description = 'Show top lines from each file')
5 parser.add_argument('filenames', nargs='+')
6 parser.add_argument('-l', '--lines', type=int, default=10)
7 args = parser.parse_args()
8 print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## Error Output Redirection and Program Termination

The sys module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
sys.stderr.write('Warning, log file not found starting a new one\n') Warning, log file not
found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## String Pattern Matching

The re module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
import re
re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest') ['foot', 'fell', 'fastest']
re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat') 'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
'tea for too'.replace('too', 'two') 'tea for two'
```

## Mathematics

The math module gives access to the underlying C library functions for floating point math:

```
import math math.cos(math.pi / 4) 0.70710678118654757 math.log(1024, 2) 10.0
```

The random module provides tools for making random selections:

```
import random random.choice(['apple', 'pear', 'banana']) 'apple'
random.sample(range(100), 10) # sampling without replacement [30, 83, 16, 4, 8, 81, 41,
50, 18, 33] random.random() # random float 0.17970987693706186
random.randrange(6) # random integer chosen from range(6) 4
```

The statistics module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
import statistics data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5] statistics.mean(data)
1.6071428571428572 statistics.median(data) 1.25 statistics.variance(data)
1.3720238095238095
```

The SciPy project [<<https://scipy.org>>(%3C<https://scipy.org>>); has many other modules for numerical computations.

## Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib.request` for retrieving data from URLs and `smtplib` for sending mail:

```
from urllib.request import urlopen with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response: ... for line in response: ... line = line.decode('utf-8') # Decoding the binary data to text. ... if 'EST' in line or 'EDT' in line: # look for Eastern Time ...
print(line)
```

```
1
Nov. 25, 09:43:32 PM EST
2
3 >>> import smtplib
4 >>> server = smtplib.SMTP('localhost')
5 >>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
```

```
6 ... """To: jcaesar@example.org
7 ... From: soothsayer@example.org
8 ...
9 ... Beware the Ides of March.
10 ...
11 >>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware. :

### dates are easily constructed and formatted

```
from datetime import date
now = date.today()
now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec
2003 is a Tuesday on the 02 day of December.'
```

---

### dates support calendar arithmetic

```
birthday = date(1964, 7, 31)
age = now - birthday
age.days 14368
```

## Data Compression

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `Izma`, `zipfile` and `tarfile`. :

```
import zlib
s = b'witch which has which witches wrist watch'
len(s) 41
t = zlib.compress(s)
len(t) 37
zlib.decompress(t)
b'witch which has which witches wrist
watch'
zlib.crc32(s) 226805979
```

## Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates a modest performance advantage:

```
| | | from timeit import Timer
Timer('t=a; a=b; b=t', 'a=1; b=2').timeit() 0.57535828626024577
Timer('a,b = b,a', 'a=1; b=2').timeit() 0.54962537085770791
```

In contrast to `timeit`'s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.

## Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the `doctest` module to make sure the code remains true to the documentation:

```
1 def average(values):
2 """Computes the arithmetic mean of a list of numbers.
3
4 >>> print(average([20, 30, 70]))
5 40.0
6 """
7 return sum(values) / len(values)
8
9 import doctest
10 doctest.testmod() # automatically validate the embedded tests
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
1 import unittest
2
3 class TestStatisticalFunctions(unittest.TestCase):
4
5 def test_average(self):
6 self.assertEqual(average([20, 30, 70]), 40.0)
7 self.assertEqual(round(average([1, 5, 7])), 1), 4.3)
8 with self.assertRaises(ZeroDivisionError):
9 average([])
10 with self.assertRaises(TypeError):
11 average(20, 30, 70)
12
13 unittest.main() # Calling from the command line invokes all tests
```

## Batteries Included

Python has a "batteries included" philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpclib` and `xmlrpclib` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the `email` package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `json` package provides robust support for parsing this popular data interchange format. The `csv` module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the `xml.etree.ElementTree`, `xml.dom` and `xml.sax` packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The `sqlite3` module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

# Data Structures

<<<<< HEAD



**cs-unit-1-sprint-1-module-1-basic-string-operations-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-string-operations-1#terminal.py>

=====

e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

An example that uses most of the list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana'] fruits.count('apple') 2
fruits.count('tangerine') 0 fruits.index('banana') 3 fruits.index('banana', 4) # Find next
banana starting at position 4 6 fruits.reverse() fruits ['banana', 'apple', 'kiwi', 'banana',
'pear', 'apple', 'orange'] fruits.append('grape') fruits ['banana', 'apple', 'kiwi', 'banana',
'pear', 'apple', 'orange', 'grape'] fruits.sort() fruits ['apple', 'apple', 'banana', 'banana',
'grape', 'kiwi', 'orange', 'pear'] fruits.pop() 'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

## Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use append. To retrieve an item from the top of the stack, use pop without an explicit index. For example:

```
stack = [3, 4, 5] stack.append(6) stack.append(7) stack [3, 4, 5, 6, 7] stack.pop() 7 stack
[3, 4, 5, 6] stack.pop() 6 stack.pop() 5 stack [3, 4]
```

## Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

```
from collections import deque queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives queue.append("Graham") # Graham arrives
queue.popleft() # The first to arrive now leaves 'Eric' queue.popleft() # The second to
arrive now leaves 'John' queue # Remaining queue in order of arrival deque(['Michael',
'Terry', 'Graham'])
```

## List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
squares = [] for x in range(10): ... squares.append(x**2) ... squares [0, 1, 4, 9, 16, 25, 36,
49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a !for clause, then zero or more !for or !if clauses. The result will be a new list resulting from evaluating the expression in the context of the !for and !if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y] [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
combs = [] for x in [1,2,3]: ... for y in [3,1,4]: ... if x != y: ... combs.append((x, y)) ... combs [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the for and if statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

:

```
vec = [-4, -2, 0, 2, 4]
```

## create a new list with the values doubled

```
[x*2 for x in vec] [-8, -4, 0, 4, 8]
```

## **filter the list to exclude negative numbers**

```
[x for x in vec if x >= 0] [0, 2, 4]
```

---

## **apply a function to all the elements**

```
[abs(x) for x in vec] [4, 2, 0, 2, 4]
```

---

## **call a method on each element**

```
freshfruit = [' banana', ' loganberry ', 'passion fruit '] [weapon.strip() for weapon in
freshfruit] ['banana', 'loganberry', 'passion fruit']
```

---

## **create a list of 2-tuples like (number, square)**

```
[(x, x**2) for x in range(6)] [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

---

## **the tuple must be parenthesized, otherwise an error is raised**

```
[x, x2 for x in range(6)] File "", line 1, in [x, x2 for x in range(6)] ^ SyntaxError: invalid
syntax
```

---

## **flatten a list using a listcomp with two 'for'**

```
vec = [[1,2,3], [4,5,6], [7,8,9]] [num for elem in vec for num in elem] [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
| | | from math import pi [str(round(pi, i)) for i in range(1, 6)] ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
| | | matrix = [... [1, 2, 3, 4], ... [5, 6, 7, 8], ... [9, 10, 11, 12], ...]
```

The following list comprehension will transpose rows and columns:

```
| | | [[row[i] for row in matrix] for i in range(4)] [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the for that follows it, so this example is equivalent to:

```
| | | transposed = [] for i in range(4): ... transposed.append([row[i] for row in matrix]) ...
| | | transposed [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
| | | transposed = [] for i in range(4): ... # the following 3 lines implement the nested listcomp
| | | ... transposed_row = [] ... for row in matrix: ... transposed_row.append(row[i]) ...
| | | transposed.append(transposed_row) ... transposed [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The zip function would do a great job for this use case:

```
| | | list(zip(*matrix)) [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See tut-unpacking-arguments for details on the asterisk in this line.

## The !del statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop` method which returns a value. The `!del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
a = [-1, 1, 66.25, 333, 333, 1234.5] del a[0] a [1, 66.25, 333, 333, 1234.5] del a[2:4] a [1, 66.25, 1234.5] del a[:] a []
```

`del` can also be used to delete entire variables:

```
del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.

## Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see `typesseq`). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
t = 12345, 54321, 'hello!' t[0] 12345 t (12345, 54321, 'hello!')
```

## Tuples may be nested:

```
... u = t, (1, 2, 3, 4, 5)
```

```
u ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

## Tuples are immutable:

```
1 ... t[0] = 88888
2 Traceback (most recent call last):
3 File "<stdin>", line 1, in <module>
4 TypeError: 'tuple' object does not support item assignment
```

## but they can contain mutable objects:

```
... v = ([1, 2, 3], [3, 2, 1])
```

```
v ([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of namedtuples \). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
empty = () singleton = 'hello', # <- note trailing comma len(empty) 0 len(singleton) 1
singleton ('hello')
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

## Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'} print(basket) # show that
duplicates have been removed {'orange', 'banana', 'pear', 'apple'} 'orange' in basket # fast
membership testing True 'crabgrass' in basket False
```

---

## Demonstrate set operations on unique letters from two words

```
...
```

```
a = set('abracadabra') b = set('alacazam') a # unique letters in a {'a', 'r', 'b', 'c', 'd'} a - b #
letters in a but not in b {'r', 'd', 'b'} a | b # letters in a or b or both {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
a & b # letters in both a and b {'a', 'c'} a ^ b # letters in a or b but not both {'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions \, set comprehensions are also supported:

```
a = {x for x in 'abracadabra' if x not in 'abc'} a {'r', 'd'}
```

## Dictionaries

Another useful data type built into Python is the *dictionary* (see typesmapping). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append` and `extend`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {} . Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del` . If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
tel = {'jack': 4098, 'sape': 4139} tel['guido'] = 4127 tel {'jack': 4098, 'sape': 4139, 'guido':
4127} tel['jack'] 4098 del tel['sape'] tel['irv'] = 4127 tel {'jack': 4098, 'guido': 4127, 'irv':
```

```
4127} list(tel) ['jack', 'guido', 'irv'] sorted(tel) ['guido', 'irv', 'jack'] 'guido' in tel True 'jack' not in tel False
```

The dict constructor builds dictionaries directly from sequences of key-value pairs:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)]) {'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
{x: x**2 for x in (2, 4, 6)} {2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098) {'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items method. :

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'} for k, v in knights.items(): ... print(k, v)
... gallahad the pure robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate function. :

```
for i, v in enumerate(['tic', 'tac', 'toe']): ... print(i, v) ... 0 tic 1 tac 2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the zip function. :

```
questions = ['name', 'quest', 'favorite color'] answers = ['lancelot', 'the holy grail', 'blue'] for q, a in zip(questions, answers): ... print('What is your {0}? It is {1}'.format(q, a)) ... What is your name? It is lancelot. What is your quest? It is the holy grail. What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed function. :

```
| | | for i in reversed(range(1, 10, 2)): ... print(i) ... 9 7 5 3 1
```

To loop over a sequence in sorted order, use the sorted function which returns a new sorted list while leaving the source unaltered. :

```
| | | basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana'] for i in sorted(basket): ...
| | | print(i) ... apple apple banana orange orange pear
```

Using set on a sequence eliminates duplicate elements. The use of sorted in combination with set over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order. :

```
| | | basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana'] for f in sorted(set(basket)): ...
| | | print(f) ... apple banana orange pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead. :

```
| | | import math raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
| | | filtered_data = [] for value in raw_data: ... if not math.isnan(value): ...
| | | filtered_data.append(value) ... filtered_data [56.2, 51.7, 55.3, 52.5, 47.8]
```

## More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `c` are true but `B` is false, `A and B and C` does not evaluate the expression `c`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example :

```
string1, string2, string3 = ", 'Trondheim', 'Hammer Dance' non_null = string1 or string2 or
string3 non_null 'Trondheim'
```

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the walrus operator `:=`. This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

## Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
1 (1, 2, 3) < (1, 2, 4)
2 [1, 2, 3] < [1, 2, 4]
3 'ABC' < 'C' < 'Pascal' < 'Python'
4 (1, 2, 3, 4) < (1, 2, 4)
```

```
5 (1, 2) < (1, 2, -1)
6 (1, 2, 3) == (1.0, 2.0, 3.0)
7 (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a')), 4)
```

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.

## Footnotes

chaining, such as ``d->insert("a")->remove("b")->sort();``.

# Links

<<<<< HEAD

## Table of contents

=====

---

## Table of contents

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

□ = External resource, □ = Beginner topic, □ = Advanced topic

### 1. About Python

- Overview: What is Python (□, □)
- Design philosophy: The Zen of Python (□)
- Style guide: Style Guide for Python Code (□, □)
- Data model: Data model (□, □)
- Standard library: The Python Standard Library (□, □)
- Built-in functions: Built-in Functions (□)

### 2. Syntax

- Variable: Built-in literals (□)
- Expression: Numeric operations (□)
- Conditional: if | if-else | if-elif-else (□)
- Loop: for-loop | while-loop (□)
- Function: def | lambda (□)

### 3. Data Structures

- List: List operations (□)
- Tuple: Tuple operations
- Set: Set operations
- Dict: Dictionary operations (□)
- Comprehension: list | tuple | set | dict
- String: String operations (□)
- Deque: deque (□)

- Time complexity: cPython operations (□, □)

#### 4. Classes

- Basic class: Basic definition (□)
- Abstract class: Abstract definition
- Exception class: Exception definition
- Iterator class: Iterator definition | yield (□)

#### 5. Advanced

- Decorator: Decorator definition | wraps (□)
- Context manager: Context managers (□)
- Method resolution order: mro (□)
- Mixin: Mixin definition (□)
- Metaclass: Metaclass definition (□)
- Thread: ThreadPoolExecutor (□)
- Asyncio: async | await (□)
- Weak reference: weakref (□)
- Benchmark: cProfile | pstats (□)
- Mocking: MagicMock | PropertyMock | patch (□)
- Regular expression: search | findall | match | fullmatch (□)
- Data format: json | xml | csv (□)
- Datetime: datetime | timezone (□)

---

## Additional resources

□ = Interview resource, □ = Code samples, □ = Project ideas

## GitHub repositories

Keep learning by reading from other well-regarded resources.

- TheAlgorithms/Python (□, □)
- faif/python-patterns (□, □)
- geekcomputers/Python (□)
- trekhleb/homemade-machine-learning (□)
- karan/Projects (□)
- MunGell/awesome-for-beginners (□)
- vinta/awesome-python

- academic/awesome-datascience
- josephmisiti/awesome-machine-learning
- ZuzooVn/machine-learning-for-software-engineers

## Interactive practice

Keep practicing so that your coding skills don't get rusty.

- leetcode.com (□)
- hackerrank.com (□)
- kaggle.com (□)
- exercism.io
- projecteuler.net
- DevProjects

# Python Glossary

## Python Glossary

<<<<< HEAD

### Glossary

#### Glossary

>>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter. . . .

Can refer to:

- The default Python prompt of the interactive shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- The `Ellipsis` built-in constant.

### 2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See [2to3 - Automated Python 2 to 3 code translation](#).

Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module),

import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.`annotation`

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a type hint.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See variable annotation, function annotation, [PEP 484](#) and [PEP 526](#), which describe this functionality.

A value passed to a function (or method) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
1 complex(real=3, imag=5)
2 complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
1 complex(3, 5)
2 complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry, the FAQ question on the difference between arguments and parameters, and [PEP 362](#).asynchronous context manager

An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by [PEP 492](#).asynchronous generator

A function which returns an asynchronous generator iterator. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.asynchronous generator iterator

An object created by a asynchronous generator function.

This is an asynchronous iterator which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See [PEP 492](#) and [PEP 525](#).asynchronous iterable

An object, that can be used in an `async for` statement. Must return an asynchronous iterator from its `__aiter__()` method. Introduced by [PEP 492](#).asynchronous iterator

An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an awaitable object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by [PEP 492](#).attribute

A value associated with an object which is referenced by name using dotted expressions. For example, if an object `o` has an attribute `a` it would be referenced as `o.a.awaitable`

An object that can be used in an `await` expression. Can be a coroutine or an object with an `__await__()` method. See also [PEP 492.BDFL](#)

Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.binary file

A file object able to read and write bytes-like objects. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also [text file](#) for a file object able to read and write `str` objects.bytes-like object

An object that supports the Buffer Protocol and can export a C-contiguous buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as “read-write bytes-like objects”. Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects (“read-only bytes-like objects”); examples of these include `bytes` and a `memoryview` of a `bytes` object.bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.callback

A subroutine function which is passed as an argument to be executed at some point in the future.class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).coercion

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.context manager

An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.context variable

A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different

points. They can be implemented with the `async def` statement. See also [PEP 492](#).coroutine function

A function which returns a coroutine object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords.

These were introduced by [PEP 492](#).CPython

The canonical implementation of the Python programming language, as distributed on [python.org](http://python.org). The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
1 def f(...):
2 ...
3 f = staticmethod(f)
4
5 @staticmethod
6 def f(...):
7 ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for [function definitions](#) and [class definitions](#) for more about [decorators.descriptor](#)

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see [Implementing Descriptors](#) or the [Descriptor How To Guide](#).

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.[dictionary comprehension](#)

A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See [Displays for lists, sets and dictionaries](#).

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See [Dictionary view objects](#).

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language

constructs are expressions. There are also statements which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

A module written in C or C++, using Python's C API to interact with the core and with user code.`f-string`

String literals prefixed with `'f'` or `'F'` are commonly called "f-strings" which is short for formatted string literals. See also [PEP 498](#).

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

A synonym for `file` object.`finder`

An object that tries to find the loader for a module that is being imported.

Since Python 3.3, there are two types of finder: meta path finders for use with `sys.meta_path`, and path entry finders for use with `sys.path_hooks`.

See [PEP 302](#), [PEP 420](#) and [PEP 451](#) for much more detail.

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*.

See [PEP 238](#).

A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also `parameter`, `method`, and the `Function definitions` section.

`function annotation`

Function annotations are usually used for type hints: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
1 def sum_two_numbers(a: int, b: int) -> int:
2 return a + b
```

Function annotation syntax is explained in section [Function definitions](#).

See variable annotation and [PEP 484](#), which describe this functionality.`__future__`

A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:>>>

```
1 >>> import __future__
2 >>> __future__.division
3 _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a generator function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending `try`-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

An expression that returns an iterator. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:>>>

```
1 >>> sum(i*i for i in range(10)) # sum of squares 0, 1, 4, ... 81
2 285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the single dispatch glossary entry, the `functools.singledispatch()` decorator, and [PEP 443](#).generic type

A type that can be parameterized; typically a container like `list`. Used for type hints and annotations.

See [PEP 483](#) for more details, and `typing` or generic alias type for its uses.GIL

See global interpreter lock.global interpreter lock

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.`hash-based pyc`

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See `Cached bytecode invalidation.hashable`

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.`IDLE`

An Integrated Development Environment for Python. `IDLE` is a basic editor and interpreter environment which ships with the standard distribution of Python.`immutable`

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.`import path`

A list of locations (or path entries) that are searched by the path based finder for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package’s `__path__` attribute.`importing`

The process by which Python code in one module is made available to Python code in another module.`importer`

An object that both finds and loads a module; both a finder and loader object.`interactive`

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also `interactive.interpreter.shutdown`

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the garbage collector. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, file objects, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements Sequence semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also `iterator`, `sequence`, and `generator.iterator`.

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator

object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in [Iterator Types](#).`key` function

A `key` function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept `key` functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a `key` function. For example, the `str.lower()` method can serve as a `key` function for case insensitive sorts. Alternatively, a `key` function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three `key` function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the [Sorting HOW TO](#) for examples of how to create and use `key` functions.`.keyword argument`

See `argument.lambda`

An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a `lambda` function is

```
lambda [parameters]: expression LBYL
```

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code,

```
if key in mapping: return mapping[key]
```

can fail if another thread removes `key` from `mapping` after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is  $O(1)$ .

A compact way to process all or part of the elements in a sequence and return a list with the results. 

```
result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]
```

 generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a finder. See [PEP 302](#) for details and `importlib.abc.Loader` for an abstract base class.

magic method

An informal synonym for special method `__getitem__`.

A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from path entry finders.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in [Metaclasses](#).

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See [function](#) and [nested scope](#).

Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

See also [package](#).

A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec.MRO`

See [method resolution order](#).

Mutable objects can change their value but keep their `id()`. See also [immutable](#).

The term “named tuple” applies to any type or class that inherits from `tuple` and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
1 >>> sys.float_info[1] # indexed access
2 1024
3 >>> sys.float_info.max_exp # named field access
4 1024
5 >>> isinstance(sys.float_info, tuple) # kind of tuple
6 True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function

`collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces.

Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.namespace package

A **PEP 420** package which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a regular package because they have no `__init__.py` file.

See also `module.nested scope`

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.package

A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also `regular package` and `namespace package.parameter`

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw\_only1* and *kw\_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the Function definitions section, and **PEP 362**.path entry

A single location on the import path which the path based finder consults to find modules for importing.path entry finder

A finder returned by a callable on `sys.path_hooks` (i.e. a path entry hook) which knows how to locate modules given a path entry.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.path entry hook

A callable on the `sys.path_hook` list which returns a path entry finder if it knows how to find modules on a specific path entry.path based finder

One of the default meta path finders which searches an import path for modules.path-like object

An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in [PEP 420](#).

See [argument.provisional API](#)

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a “solution of last resort” - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See [PEP 411](#) for more details.

See [provisional API](#).[Python 3000](#)

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.[Pythonic](#)

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
1 for i in range(len(food)):
2 print(food[i])
```

As opposed to the cleaner, [Pythonic](#) method:

```
1 for piece in food:
2 print(piece)
```

## qualified name

A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in [PEP 3155](#). For top-level functions and classes, the qualified name is the same as the object’s name:>>>

```
1 >>> class C:
2 ... class D:
3 ... def meth(self):
4 ... pass
5 ...
6 >>> C.__qualname__
7 'C'
8 >>> C.D.__qualname__
9 'C.D'
10 >>> C.D.meth.__qualname__
11 'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text` :>>>

```
1 >>> import email.mime.text
2 >>> email.mime.text.__name__
3 'email.mime.text'
```

## reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

A traditional package, such as a directory containing an `__init__.py` file.

See also namespace package.`__slots__`

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See Displays for lists, sets and dictionaries.

A form of generic function dispatch where the implementation is chosen based on the type of a single argument.

An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in Special method names.

A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`.

A codec which encodes Unicode strings to bytes.

A file object able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically. Examples of text files are files opened in text mode ( `'r'` or `'w'` ), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also [binary file](#) for a file object able to read and write bytes-like objects.[triple-quoted string](#)

A string which is bound by three instances of either a quotation mark ("") or an apostrophe (''). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.[type](#)

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.[type alias](#)

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying type hints. For example:

```
1 def remove_gray_shades(
2 colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
3 pass
```

could be made more readable like this:

```
1 Color = tuple[int, int, int]
2
3 def remove_gray_shades(colors: list[Color]) -> list[Color]:
4 pass
```

See `typing` and [PEP 484](#), which describe this functionality.[type hint](#)

An annotation that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.

An annotation of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
1 class C:
2 field: 'annotation'
```

Variable annotations are usually used for type hints: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section [Annotated assignment statements](#).

See function annotation, **PEP 484** and **PEP 526**, which describe this functionality.

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv` .virtual machine

A computer defined entirely in software. Python's virtual machine executes the bytecode emitted by the bytecode compiler.Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “ `import this` ” at the interactive prompt.

## **Binary Distribution**

A specific kind of Built Distribution that contains compiled extensions.Built Distribution

A Distribution format containing files and metadata that only need to be moved to the correct location on the target system, to be installed. Wheel is such a format, whereas distutil's Source Distribution is not, in that it requires a build step before it can be installed. This format does not imply that Python files have to be precompiled (Wheel intentionally does not include compiled Python files).Distribution Package

A versioned archive file that contains Python packages, modules, and other resource files that are used to distribute a Release. The archive file is what an end-user will download from the internet and install.

A distribution package is more commonly referred to with the single words “package” or “distribution”, but this guide may use the expanded term when more clarity is needed to prevent confusion with an Import Package (which is also commonly called a “package”) or another kind of distribution (e.g. a Linux distribution or the Python language distribution), which are often referred to with the single term “distribution”.Egg

A Built Distribution format introduced by setuptools, which is being replaced by Wheel. For details, see ` The Internal Structure of Python Eggs and Python EggsExtension Module

A Module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions.Known Good Set (KGS)

A set of distributions at specified versions which are compatible with each other. Typically a test suite will be run which passes all tests before a specific set of packages is declared a

known good set. This term is commonly used by frameworks and toolkits which are comprised of multiple individual distributions.

**Import Package**  
A **Python module** which can contain other modules or recursively, other packages.

**An import package** is more commonly referred to with the single word “package”, but this guide will use the expanded term when more clarity is needed to prevent confusion with a Distribution Package which is also commonly called a “package”.

**Module**  
The basic unit of code reusability in Python, existing in one of two types: Pure Module, or Extension Module.

**Package Index**  
A repository of distributions with a web interface to automate package discovery and consumption.

**Per Project Index**  
A private or other non-canonical Package Index indicated by a specific Project as the index preferred or required to resolve dependencies of that project.

**Project**  
A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof that is intended to be packaged into a Distribution.

Since most projects create Distributions using either **PEP 518** build-system, distutils or setuptools, another practical way to define projects currently is something that contains a `pyproject.toml`, `setup.py`, or `setup.cfg` file at the root of the project source directory.

**Distribution**  
Python projects must have unique names, which are registered on PyPI. Each project will then contain one or more Releases, and each release may comprise one or more distributions.

**Pure Module**  
Note that there is a strong convention to name a project after the name of the package that is imported to run that project. However, this doesn’t have to hold true. It’s possible to install a distribution from the project ‘foo’ and have it provide a package importable only as ‘bar’.

**Python Packaging Authority (PyPA)**  
A Module written in Python and contained in a single `.py` file (and possibly associated `.pyc` and/or `.pyo` files).

**PyPA**  
PyPA is a working group that maintains many of the relevant projects in Python packaging. They maintain a site at <https://www.pypa.io>, host projects on GitHub and Bitbucket, and

discuss issues on the distutils-sig mailing list and the Python Discourse forum.Python Package Index (PyPI)

PyPI is the default Package Index for the Python community. It is open to all Python developers to consume and distribute their distributions.pypi.org

pypi.org is the domain name for the Python Package Index (PyPI). It replaced the legacy index domain name, `pypi.python.org`, in 2017. It is powered by Warehouse.pyproject.toml

The tool-agnostic Project specification file. Defined in [PEP 518](#).Release

A snapshot of a Project at a particular point in time, denoted by a version identifier.

Making a release may entail the publishing of multiple Distributions. For example, if version 1.0 of a project was released, it could be available in both a source distribution format and a Windows installer distribution format.Requirement

A specification for a package to be installed. pip, the PYPA recommended installer, allows various forms of specification that can all be considered a “requirement”. For more information, see the pip install reference.Requirement Specifier

A format used by pip to install packages from a Package Index. For an EBNF diagram of the format, see the `pkg_resources.Requirement` entry in the setuptools docs. For example, “`foo>=1.3`” is a requirement specifier, where “`foo`” is the project name, and the “`>=1.3`” portion is the Version SpecifierRequirements File

A file containing a list of Requirements that can be installed using pip. For more information, see the pip docs on Requirements Files.`setup.py``setup.cfg`

The project specification files for distutils and setuptools. See also `pyproject.toml`.Source Archive

An archive containing the raw source code for a Release, prior to creation of a Source Distribution or Built Distribution.Source Distribution (or “`sdist`”)

A distribution format (usually generated using `python setup.py sdist`) that provides metadata and the essential source files needed for installing by a tool like pip, or for generating a Built Distribution.System Package

A package provided in a format native to the operating system, e.g. an rpm or dpkg file.**Version Specifier**

The version component of a Requirement Specifier. For example, the “ $\geq 1.3$ ” portion of “`foo $\geq 1.3$` ”. **PEP 440** contains a **full specification** of the specifiers that Python packaging currently supports. Support for PEP440 was implemented in `setuptools` v8.0 and `pip` v6.0.

[Virtual Environment](#)

An isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide. For more information, see the section on [Creating Virtual Environments](#).

**Wheel**  
A Built Distribution format introduced by **PEP 427**, which is intended to replace the Egg format. Wheel is currently supported by `pip`.

---

**A collection of distributions available for importing. These are the distributions that are on the `sys.path` variable. At most, one Distribution for a project is possible in a working set.**

 e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

# Repositories



**GitHub -  
bgoonz/DATA\_STRUC\_PYTHON\_NOTES**

[https://github.com/bgoonz/DATA\\_STRUC\\_PYTHON\\_NOTES](https://github.com/bgoonz/DATA_STRUC_PYTHON_NOTES)



**Pull requests · bgoonz/The-Algorithms**

<https://github.com/bgoonz/The-Algorithms/pulls>



**GitHub - bgoonz/python-playground-embed**

<https://github.com/bgoonz/python-playground-embed>



**GitHub - bgoonz/python-practice-notes**

<https://github.com/bgoonz/python-practice-notes>



**GitHub - bgoonz/python-scripts**

<https://github.com/bgoonz/python-scripts>



**GitHub - bgoonz/PYTHON\_PRAC**

[https://github.com/bgoonz/PYTHON\\_PRAC](https://github.com/bgoonz/PYTHON_PRAC)

# Problems w Solutions

# Python Cheat Sheet



[python-cheat-sheet.md](#)

<https://gist.github.com/bgoonz/999163a278b987fe47fb247fd4d66904#file-python-cheat-sheet-md>



[python-cheat-sheet.md](#)

<https://gist.github.com/bgoonz/999163a278b987fe47fb247fd4d66904#file-python-cheat-sheet-md>



[python-cheatsheet.py](#)

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

Embeds

# Trinket

## Trinket

First Tab

Second Tab

```
{% embed url="https://trinket.io/features/python3" caption=""%}
```



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ds-algo-forked-43ggx?file=/index.html>

# Repl.it

## Repl.it



PYTHON\_PRAC

<https://replit.com/@bgoonz/PYTHONPRAC>



BG Learn Python-2

<https://replit.com/@bgoonz/BG-Learn-Python-2#main.py>



cs-unit-1-sprint-1-module-1-loops-1

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-loops-1>



cs-unit-1-sprint-1-module-1-functions-1

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-functions-1#main.py>



python practice

<https://replit.com/@bgoonz/python-practice#basics.py>



python practice exercises

<https://replit.com/@bgoonz/python-practice-exercises#main.py>



cs-unit-1-sprint-1-module-1-basic-types-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-types-2#main.py>



**cs-unit-1-sprint-1-module-1-white-space-2**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-white-space-2>



**cs-unit-1-sprint-1-module-1-functions-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-functions-1>



**python practice-3**

<https://replit.com/@bgoonz/python-practice-3#basics.py>



**problems-w/o-solutions-1**

<https://replit.com/@bgoonz/problems-witho-solutions-1>

w/o solutions



**py-prac-medium-1**

<https://replit.com/@bgoonz/py-prac-medium-1#prac1.py>

With Solutions:



**cs-unit-1-sprint-2-module-3-stack-implementation-linked-li-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-stack-implementation-linked-li-1#main.py>



**cs-unit-1-sprint-2-module-3-stack-implementation-array-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-stack-implementation-array-1#main.py>



**cs-unit-1-sprint-2-module-3-queue-with-linked-list-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-queue-with-linked-list-1#main.py>



**cs-unit-1-sprint-4-module-2-hash-table-collision-resoluti-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-4-module-2-hash-table-collision-resoluti-1#main.py>



**cs-unit-1-sprint-4-module-1-hash-table-class-implementati-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-4-module-1-hash-table-class-implementati-1>



**cs-unit-1-sprint-1-module-1-list-comprehensions-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-list-comprehensions-1>



**cs-unit-1-sprint-1-module-2-space-complexity-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-2-space-complexity-1>



**cs-unit-1-sprint-1-module-1-conditional-expressions-1**

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-conditional-expressions-1>



**awesome-python-1**

<https://replit.com/@bgoonz/awesome-python-1#README.md>

## Keypress Action

Ctrl + A Go to the beginning of the line you are currently typing on

Ctrl + E Go to the end of the line you are currently typing on

Ctrl + L Clears the Screen, similar to the clear command

Ctrl + U Clears the line before the cursor position. If you are at the end of the line, clears the entire line.

Ctrl + H Same as backspace

---

Ctrl + R	Let's you search through previously used commands
Ctrl + C	Kill whatever you are running
Ctrl + D	Exit the current shell
Ctrl + Z	Puts whatever you are running into a suspended background process. fg restores it.
Ctrl + W	Delete the word before the cursor
Ctrl + K	Clear the line after the cursor
Ctrl + T	Swap the last two characters before the cursor
Tab	Auto-complete files and folder names

---

**{% embed url="https://replit.com/@bgoonz/intro-2-Python-through-projects-1#main.py" %}**

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

misc

# Python Problems & Solutions For Beginners

## Python Problems & Solutions For Beginners

Python Practice Problems

<<<<< HEAD

---

**{% embed url="https://replit.com/@bgoonz/problems-witho-solutions" %}**



problems-w/o-solutions

<https://replit.com/@bgoonz/problems-witho-solutions>

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

**Here are some other articles for reference if you need them:**

Beginners Guide To Python My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library...medium.com  
Python Study Guide for a JavaScript Programmer A guide to commands in Python from what you know in JavaScript levelup.gitconnected.com

**Here are the problems without solutions for you to practice with:**

### Problem 1

Create a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old.

The `datetime` module supplies classes for manipulating dates and times.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.datetime - Basic date and time types - Python 3.9.6 documentationOnly one concrete class, the class, is supplied by the module. The class can represent simple timezones with fixed...docs.python.org

## Problem 2

Ask the user for a number. Depending on whether the number is even or odd , print out an appropriate message to the user.

**Bonus:**

1. If the number is a multiple of 4 , print out a different message.
2. Ask the user for two numbers: one number to check (call it num) and one number to divide by (check). If check divides evenly into num, tell that to the user. If not, print a different appropriate message.

## Problem 3

Take a list and write a program that prints out all the elements of the list that are less than 5 .

Extras:

1. Instead of printing the elements one by one, make a new list that has all the elements less than 5 from this list in it and print out this new list.
2. Write this in one line of Python.
3. Ask the user for a number and return a list that contains only elements from the original list a that are smaller than that number given by the user.

## Problem 4

Create a program that asks the user for a number and then prints out a list of all the divisors of that number. (If you don't know what a divisor is, it is a number that divides evenly into another number.)

For example, 13 is a divisor of 26 because 26 / 13 has no remainder.)

## Problem 5

Take two lists, and write a program that returns a list that contains only the elements that are common between the lists (without duplicates) . Make sure your program works on two lists of different sizes.`random` - Generate pseudo-random numbers - Python 3.9.6 documentationSource code: [Lib/random.py](#) This module implements pseudo-random number generators for various distributions. For...[docs.python.org](#)

Bonus:

1. Randomly generate two lists to test this.
2. Write this in one line of Python.

## Problem 6

Ask the user for a string and print out whether this string is a `palindrome` or not. (A palindrome is a string that reads the same forwards and backwards.)

Here's 5 ways to reverse a string (courtesy of [geeksforgeeks](#))

## Problem 7

Let's say I give you a list saved in a variable: `a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]` .

Write one line of Python that takes this list `a` and makes a new list that has only the `even` elements of this list in it.

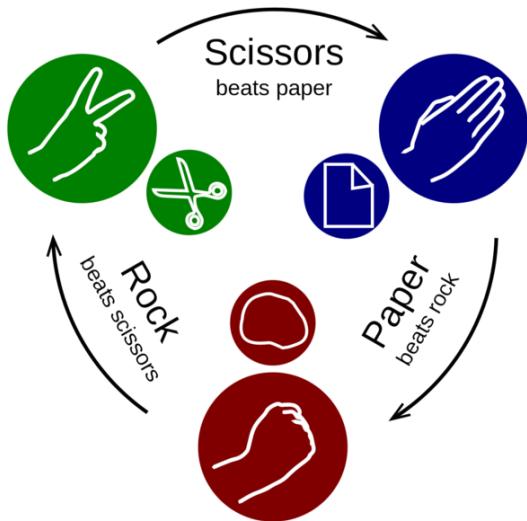
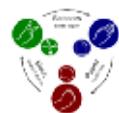
## Problem 8

Make a two-player `Rock-Paper-Scissors` game.

**Hint:**

Ask for player plays (using `input`), compare them. Print out a message of congratulations to the

winner, and ask if the players want to start a new game.



## Problem 9

Generate a random number between 1 and 100 (including 1 and 100). Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right.

### Hint:

Remember to use the user input from the very first exercise.

### Extras:

Keep the game going until the user types “exit”.

Keep track of how many guesses the user has taken, and when the game ends, print this out.

## Problem 10

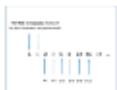
Write a program that asks the user how many Fibonacci numbers to generate and then generates them. Take this opportunity to think about how you can use functions. Make sure to ask the user to enter the number of numbers in the sequence to generate.

### Hint:

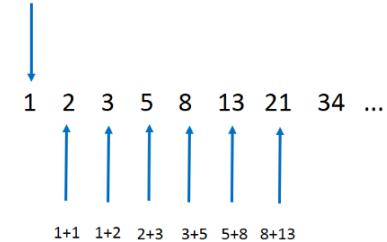
The Fibonacci sequence is a sequence of numbers where the next number in the sequence is

the sum of the previous two numbers in the sequence. The sequence looks like this:

1, 1, 2, 3, 5, 8, 13, ...



The **first** and **second** numbers in the Fibonacci sequence are 1



## Intermediate Problems:



### Problem 11

In linear algebra, a *Toeplitz matrix* is one in which the elements on any given diagonal from top left to bottom right are identical.

Here is an example:

```
1 1 2 3 4 8
2 5 1 2 3 4
3 4 5 1 2 3
4 7 4 5 1 2
```

---

Write a program to determine whether a given input is a `Toeplitz` matrix.

## Problem 12

Given a positive integer `N`, find the smallest number of steps it will take to reach `1`.

There are two kinds of permitted steps:

- → You may decrement `N` to `N - 1`.
- → If `a * b = N`, you may decrement `N` to the larger of `a` and `b`.

For example, given `100`, you can reach `1` in `5` steps with the following route:

`100 -> 10 -> 9 -> 3 -> 2 -> 1.`

## Problem 13

Consider the following scenario: there are `N` mice and `N` holes placed at integer points along a line. Given this, find a method that maps mice to holes such that the largest number of steps any mouse takes is minimized.

Each move consists of moving one mouse `one` unit to the `left` or `right`, and only `one` mouse can fit inside each hole.

For example, suppose the mice are positioned at `[1, 4, 9, 15]`, and the holes are located at `[10, -5, 0, 16]`. In this case, the best pairing would require us to send the mouse at `1` to the hole at `-5`, so our function should return `6`.

<<<<< HEAD

---

```
{% embed url="https://replit.com/@bgoonz/py-prac-medium"
%}
```



py-prac-medium

<https://replit.com/@bgoonz/py-prac-medium>

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

## **My Blog:**

Web-Dev-HubMemoization, Tabulation, and Sorting Algorithms by Example Why is looking at runtime not a reliable method of...master-bgoonz-blog.netlify.appA list of all of my articles to link to future postsYou should probably skip this one... seriously it's just for internal use!bryanguner.medium.com

---

<<<<< HEAD

||||| e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf

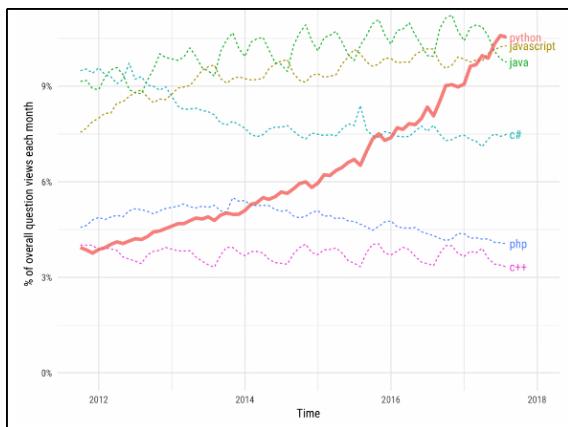
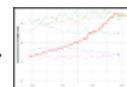
# About Python

---

## What is Python?

Python in simple words is a **High-Level Dynamic Programming Language** which is **interpreted**. Guido van Rossum , the father of Python had simple goals in mind when he was developing it, **easy looking code, readable and open source**. Python is ranked as the 3rd most prominent language followed by JavaScript and Java in a survey held in 2018 by Stack Overflow which

serves proof to it being the most growing language.



## Features of Python

Python is currently my favorite and most preferred language to work on because of its *simplicity, powerful libraries, and readability*. You may be an old school coder or may be completely new to programming, Python is the best way to get started!

Python is currently my favorite and most preferred language to work on because of its *simplicity, powerful libraries, and readability*. You may be an old school coder or may be completely new to programming, is the best way to get started!

Python provides features listed below :

- **Simplicity:** Think less of the syntax of the language and more of the code.
- **Open Source:** A powerful language and it is free for everyone to use and alter as needed.
- **Portability:** Python code can be shared and it would work the same way it was intended to. Seamless and hassle-free.
- **Being Embeddable & Extensible:** Python can have snippets of other languages inside it to perform certain functions.
- **Being Interpreted:** The worries of large memory tasks and other heavy CPU tasks are taken care of by Python itself leaving you to worry only about coding.
- **Huge amount of libraries:** Data Science Python has you covered. Web Development? Python still has you covered. Always.
- **Object Orientation:** Objects help breaking-down complex real-life problems into such that they can be coded and solved to obtain solutions.

To sum it up, Python has a **simple syntax**, is **readable**, and has **great community support**. You may now have the question, What can you do if you know Python? Well, you have a number of options to choose from.

- Data Scientist
- Machine Learning and Artificial Intelligence
- Internet of Things
- Web Development
- Data Visualization
- Automation

Now when you know that Python has such an amazing feature set, why don't we get started with the Python Basics?

---

## Jumping to the Python Basics

To get started off with the Python Basics, you need to first **install Python** in your system right? So let's do that right now! You should know that most **Linux** and **Unix** distributions these days come with a version of Python out of the box. To set yourself up, you can follow this **step-to-step guide**.

Once you are set up, you need to create your first project. Follow these steps:

- Create **Project** and enter the name and click **create**.

- Right-click on the project folder and create a **python file** using the New->File->Python File and enter the file name

You're done. You have set up your files to start coding with Python. Are you excited to start coding? Let's begin. The first and foremost, the "Hello World" program.

```
print('Hello World, Welcome to edureka!')
```

**Output:** Hello World, Welcome to edureka!

There you are, that's your first program. And you can tell by the syntax, that it is **super easy** to understand. Let us move over to comments in Python Basics.

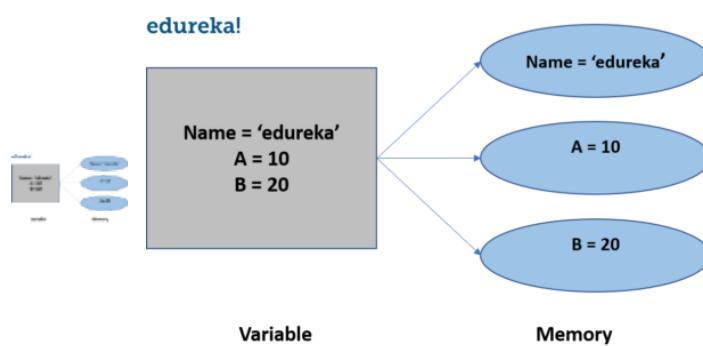
---

## Comments in Python

Single line comment in Python is done using the `#` symbol and `'''` for multi-line commenting. If you want to know more about **comments**, you can read this full-fledged guide. Once you know commenting in Python Basics, let's jump into variables in Python Basics.

---

## Variables



Variables in simple words are **memory spaces** where you can store **data or values**. But the catch here in Python is that the variables don't need to be declared before the usage as it is needed in other languages. The **data type** is **automatically assigned** to the variable. If you enter

an Integer, the data type is assigned as an Integer. You enter a string, the variable is assigned a string data type. You get the idea. This makes Python **dynamically typed language**. You use the assignment operator (=) to assign values to the variables.

```
1 a = 'Welcome to edureka!'
2 b = 123
3 c = 3.142
4 print(a, b, c)
```

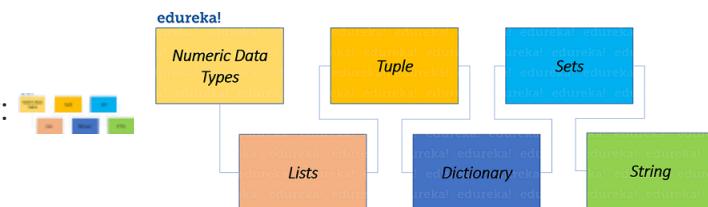
**Output:** Welcome to edureka! 123 3.142

You can see the way I have assigned the values to those variables. This is how you assign values to variables in Python. And if you are wondering, yes, you can **print multiple variables** in a single print statement. Now let us go over Data Types in Python Basics.

## Data Types in Python

Data types are basically **data** that a **language supports** such that it is helpful to define real-life data such as salaries, names of employees and so on. The possibilities are endless. The data

types are as shown below:



### Numeric Data Types

As the name suggests, this is to store numerical data types in the variables. You should know that they are **immutable**, meaning that the specific data in the variable cannot be changed.

There are 3 numerical data types :

- **Integer:** This is just as simple to say that you can store integer values in the variables. Ex :  
a = 10.
- **Float:** Float holds the real numbers and are represented by a decimal and sometimes even scientific notations with E or e indicating the power of 10 (2.5e2 = 2.5 x 10<sup>2</sup> = 250). Ex:  
10.24.

- **Complex Numbers:** These are of the form  $a + bj$ , where  $a$  and  $b$  are floats and  $J$  represents the square root of  $-1$  (which is an imaginary number). Ex:  $10+6j$ .

```
1 a = 10
2 b= 3.142
3 c = 10+6j
```

So now that you have understood the various numerical data types, you can understand converting one data type into another data type in this blog of Python Basics.

## Type Conversion

Type Conversion is the **conversion of a data type into another data type** which can be really helpful to us when we start programming to obtain solutions for our problems. Let us understand with examples.

```
1 a = 10
2 b = 3.142
3 c = 10+6j
4 print(int(b), float(a), str(c))
```

**Output:** 10.0 3 '10+6j'

You can understand, type conversion by the code snippet above. 'a' as an integer, 'b' as a float and 'c' as a complex number. You use the `float()`, `int()`, `str()` methods that are in-built in Python which helps us to convert them. **Type Conversion** can be really important when you move into real-world examples.

A simple situation could be where you need to compute the salary of the employees in a company and these should be in a float format but they are supplied to us in the string format. So to make our work easier, you just use type conversion and convert the string of salaries into float and then move forward with our work. Now let us head over to the List data type in Python Basics.

## Lists

List in simple words can be thought of as in them, i.e, **arrays** that exist in other languages but with the exception that they can have **heterogeneous elements****different data types in the same list**. Lists are **mutable**, meaning that you can change the data that is available in them.

For those of you who do not know what an array is, you can understand it by imagining a Rack that can hold data in the way you need it to. You can later access the data by calling the position in which it has been stored which is called as **Index** in a programming language. Lists are defined using either the `a=list()` method or using `a=[]` where 'a' is the name of the list.

	3.142	Hindi	135	10+3j
	A[0]	A[1]	A[2]	A[3]

You can see from the above figure, the data that is stored in the list and the index related to that data stored in the list. Note that the **Index in Python always starts with '0'**. You can now move over to the operations that are possible with Lists.

List operations are as shown below in the tabular format.



Code Snippet	Output Obtained	Operation Description
<code>a[2]</code>	135	Finds the data at index 2 and returns it
<code>a[0:3]</code>	[3.142, 'Hindi', 135]	Data from index 0 to 2 is returned as the last index mentioned is always ignored.
<code>a[3] = 'edureka!'</code>	moves 'edureka!' to index 3	The data is replaced in index 3
<code>del a[1]</code>	Deletes 'Hindi' from the list	Delete items and it does not return any item back
<code>len(a)</code>	3	Obtain the length of a variable in Python
<code>a * 2</code>	Output the list 'a' twice	If a dictionary is multiplied with a number, it is repeated that many number of times
<code>a[::-1]</code>	Output the list in the reverse order	Index starts at 0 from left to right. In reverse order, or, right to left, the index starts from -1.
<code>a.append(3)</code>	3 will be added at the end of the list	Add data at the end of the list
<code>a.extend(b)</code>	[3.142, 135, 'edureka!', 3, 2]	'b' is a list with value 2. Adds the data of the list 'b' to 'a' only. No changes are made to 'b'.
<code>a.insert(3,'hello')</code>	[3.142, 135, 'edureka!', 'hello', 3, 2]	Takes the index and the value and adds value to that index.
<code>a.remove(3.142)</code>	[135, 'edureka!', 'hello', 3, 2]	Removes the value from the list that has been passed as an argument. No value returned.
<code>a.index(135)</code>	0	Finds the element 135 and returns the index of that data
<code>a.count('hello')</code>	1	It goes through the string and finds the times it has been repeated in the list
<code>a.pop(1)</code>	'edureka!'	Pops the element in the given index and returns the element if needed.
<code>a.reverse()</code>	[2, 3, 'hello', 135]	It just reverses the list
<code>a.sort()</code>	[5, 1234, 64738]	Sorts the list based on ascending or descending order.
<code>a.clear()</code>	[]	Used to remove all the elements that are present in the list.

Now that you have understood the various list functions, let's move over to understanding Tuples in Python Basics.

## Tuples

Tuples in Python are the . That means that once you have declared the tuple, you cannot add, delete or update the tuple. Simple as that. This makes **same as lists**. Just one thing to remember, tuples are **immutable tuples much faster than Lists** since they are constant values.

Operations are similar to Lists but the ones where updating, deleting, adding is involved, those operations won't work. Tuples in Python are written `a=()` or `a=tuple()` where 'a' is the name of the tuple.

```
1 a = ('List', 'Dictionary', 'Tuple', 'Integer', 'Float')
2 print(a)
```

**Output** = ('List', 'Dictionary', 'Tuple', 'Integer', 'Float')

That basically wraps up most of the things that are needed for tuples as you would use them only in cases when you want a list that has a constant value, hence you use tuples. Let us move over to Dictionaries in Python Basics.

---

## Dictionary

Dictionary is best understood when you have a real-world example with us. The most easy and well-understood example would be of the telephone directory. Imagine the telephone directory and understand the various fields that exist in it. There is the Name, Phone, E-Mail and other fields that you can think of. Think of the *Name* as the **key** and the **name** that you enter as the **value**. Similarly, *Phone* as **key**, *entered data* as **value**. This is what a dictionary is. It is a structure that holds the **key, value** pairs.

Dictionary is written using either the `a=dict()` or using `a={}` where a is a dictionary. The key could be either a string or integer which has to be followed by a ":" and the value of that key.

```
1 MyPhoneBook = { 'Name' : ['Akash', 'Ankita'] ,
2 'Phone' : ['12345', '12354'] ,
3 'E-Mail' : ['akash@rail.com', 'ankita@rail.com']}
4 print (MyPhoneBook)
```

**Output:** { 'Name' : ['Akash', 'Ankita'], 'Phone' : ['12345', '12354'], 'E-Mail' : ['akash@rail.com','ankita@rail.com']}

## Accessing elements of the Dictionary

You can see that the keys are Name, Phone, and EMail who each have 2 values assigned to them. When you print the dictionary, the key and value are printed. Now if you wanted to obtain values only for a particular key, you can do the following. This is called accessing elements of the dictionary.

```
print(MyPhoneBook['E-Mail'])
```

**Output :** ['akash@rail.com','ankita@rail.com']

## Operations of Dictionary

You may now have a better understanding of dictionaries in Python Basics. Hence let us move

over to Sets in this blog of Python Basics.



Code Snippet	Output Obtained	Operation Description
MyPhoneBook.keys()	dict_keys(['Name', 'Phone', 'E-Mail'])	Returns all the keys of the dictionary
MyPhoneBook.values()	dict_values([('Akash', 'Ankita'), [12345, 12354], ['akash@rail.com', 'akash@rail.com'])])	Returns all the values of the dictionary
MyPhoneBook['id']=[1, 2]	{'Name': ['Akash', 'Ankita'], 'Phone': [12345, 12354], 'E-Mail': ['ankita@rail.com', 'akash@rail.com'], 'id': [1, 2]}	The new key, value pair of id is added to the dictionary
MyPhoneBook['Name'][0] = "Akki"	'Name': ['Akki', 'Ankita']	Access the list of names and change the first element.
del MyPhoneBook['id']	{'Name': ['Akash', 'Ankita'], 'Phone': [12345, 12354], 'E-Mail': ['ankita@rail.com', 'akash@rail.com']}	The key, value pair of ID has been deleted
len(MyPhoneBook)	3	3 key-value pairs in the dictionary and hence you obtain the value 3
MyPhoneBook.clear()	{}	Clear the key, value pairs and make a clear dictionary

## Sets

A set is basically an You can see that even if there are similar elements in set 'a', it will still be printed only once because **un-ordered collection of elements** or items. Elements are sets are a collection of unique elements. **unique** in the set. In Python, they are written inside **curly brackets** and **separated by commas**.

```
1 a = {1, 2, 3, 4, 4, 4}
2 b = {3, 4, 5, 6}
```

```
3 print(a,b)
```

**Output :** {1, 2, 3, 4} {3, 4, 5, 6}

## Operations in Sets

Sets are simple to understand, so let us move over to strings in Python Basics.



Code Snippet	Output Obtained	Operation Description
a   b	{1, 2, 3, 4, 5, 6}	Union operation where all the elements of the sets are combined.
a & b	{3, 4}	Intersection operation where only the elements present in both sets are selected.
a - b	{1, 2}	Difference operation where the elements present in 'a' and 'b' are deleted and remaining elements of 'a' is the result.
a ^ b	{1, 2, 5, 6}	Symmetric difference operation where the intersecting elements are deleted and the remaining elements in both sets is the result.

## Strings

Strings in Python are the most used data types, especially because they are easier for us humans to interact with. They are literally words and letters which makes sense as to how they are being used and in what context. Python hits it out of the park because it has such a powerful integration with strings. Strings are written within a **single ("") or double quotation marks ("")**. Strings are **immutable** meaning that the data in the string cannot be changed at particular indexes.

The operations of strings with Python can be shown as:

**Note: The string here I use is : mystr ="edureka! is my place"**



Code Snippet	Output Obtained	Operation Description
len(mystr)	20	Finds the length of the string
mystr.index('!')	7	Finds the index of the given character in the string
mystr.count('!')	1	Finds the count of the character passed as the parameter
mystr.upper()	EDUREKA! IS MY PLACE	Converts all the string into the upper case
mystr.split(' ')	['edureka!', 'is', 'my', 'place']	Breaks the string based on the delimiter passed as the parameter.
mystr.lower()	edureka! is my place	Converts all the strings of the string into lower case
mystr.replace(' ', ',')	edureka!,is,my,place	Replaces the string which has old value with the new value.
mystr.capitalize()	Edureka! is my place	This capitalizes the first letter of the string

These are just a few of the functions available and you can find more if you search for it.

## Splicing in Strings

Splicing is **breaking the string** into the format or the way you want to obtain it.

That basically sums up the data types in Python. I hope you have a good understanding of the same and if you have any doubts, please leave a comment and I will get back to you as soon as possible.

Now let us move over to Operators in Python Basics.

---

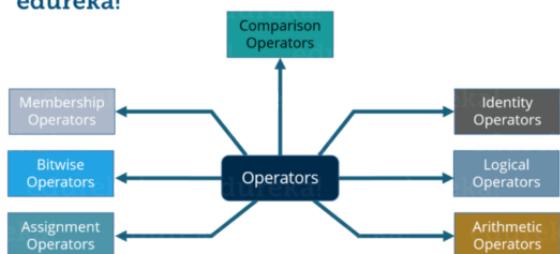
## Operators in Python

Operators are **constructs** you use to **manipulate the data** such that you can conclude some sort of solution to us. A simple example would be that if there were 2 friends having 70 rupees each, and you wanted to know the total they each had, you would add the money. In Python, you use the + operator to add the values which would sum up to 140, which is the solution to the problem.

Python has a list of operators which can be grouped as :



**edureka!**



Let us move ahead and understand each of these operators carefully.

**Note: Variables are called operands that come on the left and right of the operator. Ex :**

```
1 a=10
2 b=20
3 a+b
```

Here 'a' and 'b' are the operands and + is the operator.

## Arithmetic Operator

They are used to perform **arithmetic operations** on data.



Operator	Description
+	Adds the values of the operands
-	Subtracts the value of the right operator with the left operator
*	Multiples left operand with the right operand
/	Divides the left operand with the right operand
%	Divides the left operand with the right operand and returns the remainder
**	Performs the exponential of the left operand with the right operand

The code snippet below will help you understand it better.

```
1 a = 2
2 b = 3
3 print(a+b, a-b, a*b, a/b, a%b, a**b, end=',')
```

**Output :** 5, -1, 6, 0.6666666666666666, 2, 8

Once you have understood what the arithmetic operators are in Python Basics, let us move to assignment operators.

## Assignment Operators

As the name suggests, these are used to **assign values to the variables**. Simple as that.

The various assignment operators are :

Operator	Description
=	It is used to assign the value on the right to the variable on the left
+=	Notation for assigning the value of the addition of the left and right operand to the left operand.
-=	Notation for assigning the value of the difference of the left and right operand to the left operand.
*=	Short-hand notation for assigning the value of the product of the left and right operand to the left operand.
/=	Short-hand notation for assigning the value of the division of the left and right operand to the left operand.
%=	Short-hand notation for assigning the value of the remainder of the left and right operand to the left operand.

Let us move ahead to comparison operators in this blog of Python Basics.

## Comparison Operators

These operators are used to **bring out the relationship** between the left and right operands and derive a solution that you would need. It is as simple as to say that you use them for **comparison purposes**. The output obtained by these operators will be either true or false depending if the condition is true or not for those values.



Operator	Description
==	Find out whether the left and right operands are equal in value or not
!=	Find out whether the values of left and right operators are not equal
<	Find out whether the value of the right operand is greater than that of the left operand
>	Find out whether the value of the left operand is greater than that of the right operand
<=	Find out whether the value of the right operand is greater than or equal to that of the left operand
>=	Find out whether the value of the left operand is greater than or equal to that of the right operand

You can see the working of them in the example below :

```
1 a = 21
2 b = 10
3 if a == b:
4 print ('a is equal to b')
5 if a != b
6 print ('a is not equal to b')
7 if a < b:
8 print ('a is less than b')
9 if a > b:
10 print ('a is greater than b')
11 if a <= b:
12 print ('a is either less than or equal to b')
13 if a >= b:
14 print ('a is either greater than or equal to b')
```

### Output :

a is not equal to b  
a is greater than b  
a is either greater than or equal to b

Let us move ahead with the bitwise operators in the Python Basics.

## Bitwise Operators

To understand these operators, you need to understand the **theory of bits**. These operators are

used to **directly manipulate the bits**.

Operator	Description
&	Used to do the AND operation on individual bits of the left and right operands
	Used to do the OR operation on individual bits of the left and right operands
^	Used to do the XOR operation on individual bits of the left and right operands
~	Used to do the 1's compliment operation on individual bits of the left and right operands
<<	Used to shift the left operand by right operand times. One left shift is equivalent to multiplying by 2.
>>	Used to shift the left operand by right operand times. One right shift is equivalent to dividing by 2.

It would be better to practice this by yourself on a computer. Moving ahead with logical operators in Python Basics.

## Logical Operators

These are used to obtain a certain **logic** from the operands. We have 3 operands.

- **and** (True if both left and right operands are true)
- **or** (True if either one operand is true)
- **not** (Gives the opposite of the operand passed)

```
1 a = True
2 b = False
3 print(a and b, a or b, not a)
```

**Output:** False True False

Moving over to membership operators in Python Basics.

## Membership Operators

These are used to test whether a **particular variable** or value **exists** in either a list, dictionary, tuple, set and so on.

The operators are :

- **in** (True if the value or variable is found in the sequence)
- **not in** (True if the value is not found in the sequence)

```
1 a = [1, 2, 3, 4]
2 if 5 in a:
3 print('Yes!')
4 if 5 not in a:
5 print('No!')
```

**Output:** No!

Let us jump ahead to identity operators in Python Basics.

## Identity Operator

These operators are used to **check whether the values**, variables are **identical** or not. As simple as that.

The operators are :

- **is** (True if they are identical)
- **is not** (True if they are not identical)

```
1 a = 5
2 b = 5
3 if a is b:
4 print('Similar')
5 if a is not b:
6 print('Not Similar!')
```

That right about concludes it for the operators of Python.

---

## Namespacing and Scopes

You do remember that **everything in Python is an object**, right? Well, how does Python know what you are trying to access? Think of a situation where you have 2 functions with the same name. You would still be able to call the function you need. How is that possible? This is where namespacing comes to the rescue.

Namespacing is the system that Python uses to assign **unique names** to all the objects in our code. And if you are wondering, objects can be variables and methods. Python does namespacing by **maintaining a dictionary structure**. Where *names act as the keys* and the *object or variable acts as the values in the structure*. Now you would wonder what is a name?

Well, a is just a way that you use to **nameaccess the objects**. These names act as a reference to access the values that you assign to them.

**Example:** a=5, b='edureka!'

If I would want to access the value 'edureka!' I would simply call the variable name by 'b' and I would have access to 'edureka!'. These are names. You can even assign methods names and call them accordingly.

```
1 import math
2 square_root = math.sqrt
3 print('The square root is ',square_root(9))
```

**Output:** The root is 3.0

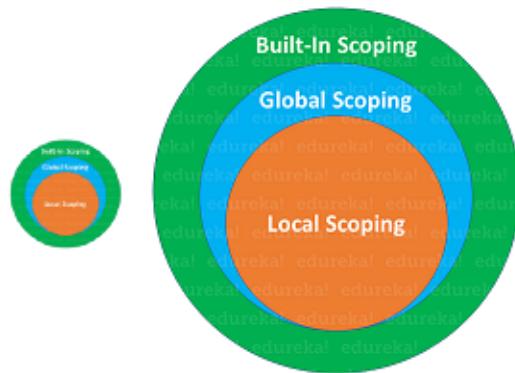
Namespacing works with scopes. **Scopes** are the *validity of a function/variable/value inside the function or class they belong to*. Python **built-in functions** namespacing **covers all the other scopes of Python**. Functions such as print() and id() etc. can be used even without any imports and be used anywhere. Below them is the **global** and **local** namespacing. Let me explain the scope and namespacing in a code snippet below :

```
1 def add():
2 x = 3
3 y = 2
4 def add2():
5 p, q, r = 3, 4, 5
6 print('Inside add2 printing sum of 3 numbers:'(p+q+r))
7 add2()
8 print('The values of p, q, r are :', p, q, r)
9 print('Inside the add printing sum of 2 numbers:'(x+y))
10 add()
```

As you can see with the code above, I have declared 2 functions with the name add() and add2(). You have the definition of the add() and you later call the method add(). Here in add() you call add2() and so you are able to get the output of 12 since 3+4+5 is 12. But as soon as you come out of add2(), the scope of p,q,r is terminated meaning that p,q,r are only accessible and available if you are in add2(). Since you are now in add(), there is no p,q,r and hence you get the error and execution stops.

You can get a better understanding of the scopes and namespaces from the figure below. The **built-in scope** covers all of Python making them *available whenever needed*. The **global scope** covers all of the *programs* that are being executed. The **local scope** covers all of the *methods* being executed in a program. That is basically what namespaces is in Python. Let us move

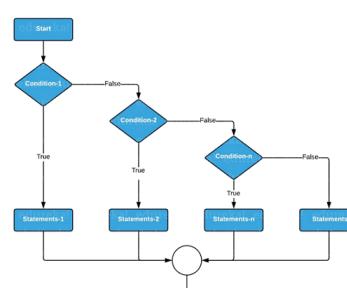
ahead with flow control in Python Basics.



## Flow Control and Conditioning in Python

You know that code runs sequentially in any language, but what if you want to **break that flow** such that you are able to **add logic and repeat certain statements** such that your code reduces and are able to obtain a **solution with lesser and smarter code**. After all, that is what coding is. Finding logic and solutions to problems and this can be done using loops in Python and

conditional statements.



Conditional statements are **executed** only if a **certain condition is met**, else it is **skipped** ahead to where the condition is satisfied. Conditional statements in Python are the **if, elif and else**.

## Syntax:

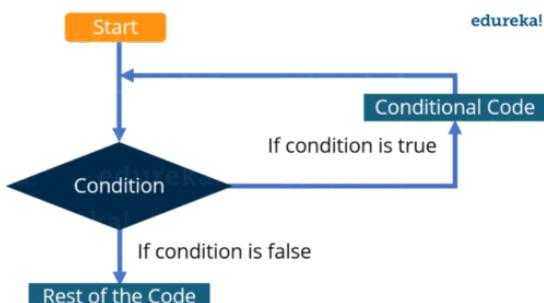
```
1 if condition:
2 statement
3 elif condition:
4 statement
5 else:
6 statement
```

This means that if a condition is met, do something. Else go through the remaining elif conditions and finally if no condition is met, execute the else block. You can even have nested if-else statements inside the if-else blocks.

```
1 a = 10
2 b = 15
3 if a == b:
4 print ('They are equal')
5 elif a > b:
6 print ('a is larger')
7 else :
8 print ('b is larger')
```

**Output:** b is larger

With conditional statements understood, let us move over to loops. You would have certain times when you would want to execute certain statements again and again to obtain a solution or you could apply some logic such that a certain similar kind of statements can be executed using only 2 to 3 lines of code. This is where you use **loops in Python**.



Loops can be divided into 2 kinds.

- **Finite:** This kind of loop works until a certain condition is met
- **Infinite:** This kind of loop works infinitely and does not stop ever.

Loops in Python or any other language have to test the condition and they can be done either before the statements or after the statements. They are called :

- **Pre-Test Loops:** Where the condition is tested first and statements are executed following that
- **Post Test Loops:** Where the statement is executed once at least and later the condition is checked.

You have 2 kinds of loops in Python:

- **for**
- **while**

Let us understand each of these loops with syntaxes and code snippets below.

**For Loops:** These loops are used to perform a **certain set of statements** for a given **condition** and continue until the condition has failed. You know the **number of times** that you need to execute the for loop.

**Syntax:**

```
for variable in range: statements
```

The code snippet is as below :

```
1 basket_of_fruits= ['apple', 'orange', 'pineapple', 'banana']
2 for fruit in basket_of_fruits:
3 print(fruit, end=',')
```

**Output:** apple, orange, pineapple, banana

This is how the for loops work in Python. Let us move ahead with the while loop in Python Basics.

**While Loops:** While loops are the **same as the for loops** with the exception that you may not know the ending condition. For loop conditions are known but the **while loop conditions** may not.

**Syntax:**

```
1 while condition:
2 statements
```

The code snippet is as :

```
1 second = 10
2 while second >= 0:
3 print(second, end='->')
4 second-=1
5 print('Blastoff!')
```

**Output :** 10->9->8->7->6->5->4->3->2->1->Blastoff!

This is how the while loop works.

You later have **nested loops** where you **embed a loop into another**. The code below should give you an idea.

```
1 count = 1
2 for i in range(10):
3 print(str(i) * i)
4 for j in range(0, i):
5 count = count+1
```

**Output :**

1

22

333

4444

55555

666666

777777

88888888

99999999

You have the first for loop which prints the string of the number. The other for loop adds the number by 1 and then these loops are executed until the condition is met. That is how for loop works. And that wraps up our session for loops and conditions. Moving ahead with file handling in Python Basics.

---

## File Handling with Python

Python has built-in functions that you can use to **work with files** such as **reading** and **writing data from or to a file**. A **file object** is returned when a file is called using the `open()` function and then you can do the operations on it such as `read`, `write`, `modify` and so on.

The flow of working with files is as follows :

- **Open** the file using the `open()` function
- Perform **operations** on the file object
- **Close** the file using the `close()` function to avoid any damage to be done with the file

### Syntax:

```
File_object = open('filename','r')
```

Where mode is the way you want to interact with the file. If you do not pass any mode variable,

the default is taken as the read mode.



Mode	Description
r	Default mode in Python. It is used to read the content from a file.
w	Used to open in write mode. If a file does not exist, it shall create a new one else truncates the contents of the present file.
x	Used to create a file. If the file exists, the operation fails
a	Open a file in append mode. If the file does not exist, then it opens a new file.
b	This reads the contents of the file in binary.
t	This reads the contents in text mode and is the default mode in Python.

### Example:

```
1 file = open('mytxt','w')
2 string = ' --Welcome to edureka!-- '
3 for i in range(5):
4 file.write(string)
5 file.close()
```

**Output:** -Welcome to edureka!- in mytxt file

You can go ahead and try more and more with files. Let's move over to the last topics of the blog. OOPS and objects and classes. Both of these are closely related.

---

## OOPS

Older programming languages were structured such that **data could be accessed by any module of the code**. This could lead to **potential security issues** that led developers to move over to **Object-Oriented Programming** that could help us emulate real-world examples into code such that better solutions could be obtained.

There are 4 concepts of OOPS which are important to understand. They are:

- **Inheritance:** Inheritance allows us to **derive attributes and methods** from the parent class and modify them as per the requirement. The simplest example can be for a car where the structure of a car is described and this class can be derived to describe sports cars, sedans and so on.
- **Encapsulation:** Encapsulation is **binding data and objects together** such that other objects and classes do not access the data. Python has private, protected and public types whose

names suggest what they do. Python uses '\_' or '\_\_' to specify private or protected keywords.

- **Polymorphism:** This allows us to have a **common interface for various types of data** that it takes. You can have similar function names with differing data passed to them.
- Abstraction can be used to **Abstraction: simplify complex reality by modeling classes** appropriate to the problem.

# Misc

Links:



[35-repositories.md](#)

<https://gist.github.com/bgoonz/9fb85e86115881a212a293c9829aade2>

# Numbers

Computers are designed to perform numerical calculations, but there are some important details about working with numbers that every programmer working with quantitative data should know. Python (and most other programming languages) distinguishes between two different types of numbers:

- Integers are called `int` values in the Python language. They can only represent whole numbers (negative, zero, or positive) that don't have a fractional component
- Real numbers are called `float` values (or *floating point values*) in the Python language. They can represent whole or fractional numbers but have some limitations.

The type of a number is evident from the way it is displayed: `int` values have no decimal point and `float` values always have a decimal point.

```
1 # Some int values
2 2
```

```
2
```

```
1 + 3
```

```
4
```

```
-12345678900000000000
```

```
-12345678900000000000
```

```
1 # Some float values
2 1.2
```

1.2

1.5 + 2

3.5

3 / 1

3.0

-12345678900000000000.0

-1.23456789e+19

When a `float` value is combined with an `int` value using some arithmetic operator, then the result is always a `float` value. In most cases, two integers combine to form another integer, but any number (`int` or `float`) divided by another will be a `float` value. Very large or very small `float` values are displayed using scientific notation.

## Float Values¶

Float values are very flexible, but they do have limits.

1. A `float` can represent extremely large and extremely small numbers. There are limits, but you will rarely encounter them.
2. A `float` only represents 15 or 16 significant digits for any number; the remaining precision is lost. This limited precision is enough for the vast majority of applications.
3. After combining `float` values with arithmetic, the last few digits may be incorrect. Small rounding errors are often confusing when first encountered.

The first limit can be observed in two ways. If the result of a computation is a very large number, then it is represented as infinite. If the result is a very small number, then it is represented as zero.

```
2e306 * 10
```

```
2e+307
```

```
2e306 * 100
```

```
inf
```

```
2e-322 / 10
```

```
2e-323
```

```
2e-322 / 100
```

```
0.0
```

The second limit can be observed by an expression that involves numbers with more than 15 significant digits. These extra digits are discarded before any arithmetic is carried out.

```
0.6666666666666666 - 0.666666666666666123456789
```

```
0.0
```

The third limit can be observed when taking the difference between two expressions that should be equivalent. For example, the expression `2 ** 0.5` computes the square root of 2, but squaring this value does not exactly recover 2.

```
2 ** 0.5
```

```
1.4142135623730951
```

```
(2 ** 0.5) * (2 ** 0.5)
```

```
2.0000000000000004
```

```
(2 ** 0.5) * (2 ** 0.5) - 2
```

4.440892098500626e-16

The final result above is `0.00000000000004440892098500626`, a number that is very close to zero. The correct answer to this arithmetic expression is 0, but a small error in the final significant digit appears very different in scientific notation. This behavior appears in almost all programming languages because it is the result of the standard way that arithmetic is carried out on computers.

Although `float` values are not always exact, they are certainly reliable and work the same way across all different kinds of computers and programming languages.

# Youtube

Here are the best YouTube channels to learn Python programming for beginners:

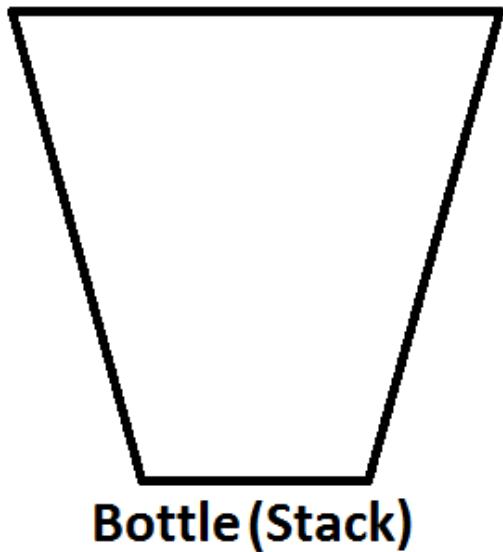
1. **Al Sweigart\*\*\*\***
2. **\*\*\*\*Anaconda Inc.\*\*\*\***
3. **\*\*\*\*Chris Hawkes\*\*\*\***
4. **\*\*\*\*Christian Thompson\*\*\*\***
5. **\*\*\*\*Clever Programmer\*\*\*\***
6. **\*\*\*\*Corey Schafer\*\*\*\***
7. **\*\*\*\*CS Dojo\*\*\*\***
8. **\*\*\*\*Derek Banas\*\*\*\***
9. **\*\*\*\*Data School\*\*\*\***
10. **\*\*\*\*freeCodeCamp\*\*\*\***
11. **\*\*\*\*Pretty Printed\*\*\*\***
12. **\*\*\*\*Programming with Mosh\*\*\*\***
13. **\*\*\*\*PyData\*\*\*\***
14. **\*\*\*\*Real Python\*\*\*\***
15. **\*\*\*\*Sentdex\*\*\*\***
16. **\*\*\*\*Socratica\*\*\*\***
17. **\*\*\*\*Telusko\*\*\*\***
18. **\*\*\*\*thenewboston\*\*\*\***
19. **\*\*\*\*Traversy Media\*\*\*\***

# How To Reverse A Stack

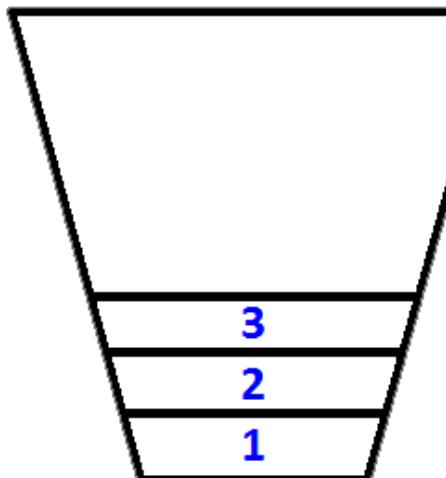
## A - How To Reverse A Stack

To understand how to construct a queue using two stacks, you should understand how to reverse a stack crystal clear. Remember how stack works, it is very similar to the dish stack on your kitchen. The last washed dish will be on the top of the clean stack, which is called as **Last In First Out (LIFO)** in computer science.

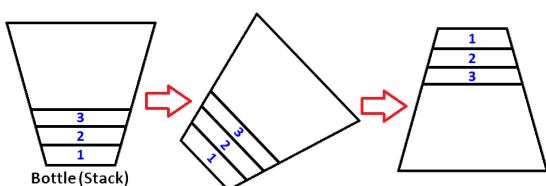
Lets imagine our stack like a bottle as below;



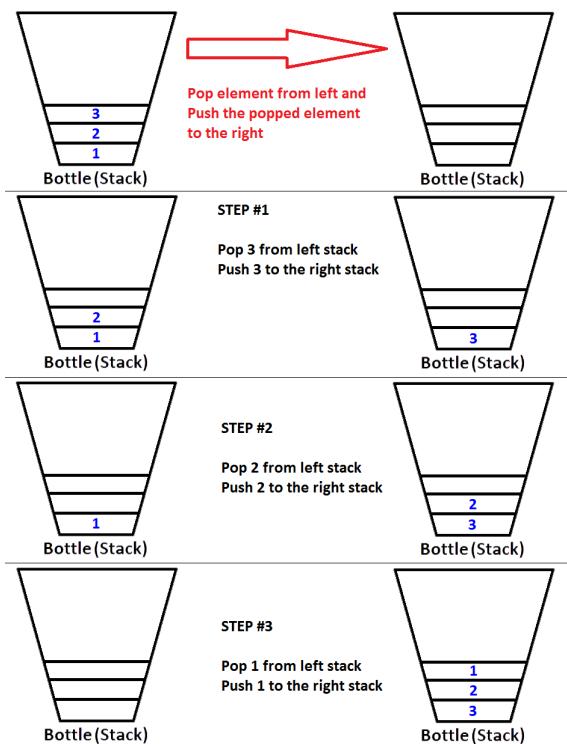
If we push integers 1,2,3 respectively, then 3 will be on the top of the stack. Because 1 will be pushed first, then 2 will be put on the top of 1. Lastly, 3 will be put on the top of the stack and latest state of our stack represented as a bottle will be as below;



Now we have our stack represented as a bottle is populated with values 3,2,1. And we want to reverse the stack so that the top element of the stack will be 1 and bottom element of the stack will be 3. What we can do ? We can take the bottle and hold it upside down so that all the values should reverse in order ?



Yes we can do that, but that's a bottle. To do the same process, we need to have a second stack that which is going to store the first stack elements in reverse order. Let's put our populated stack to the left and our new empty stack to the right. To reverse the order of the elements, we are going to pop each element from left stack, and push them to the right stack. You can see what happens as we do so on the image below;



So we know how to reverse a stack.

---

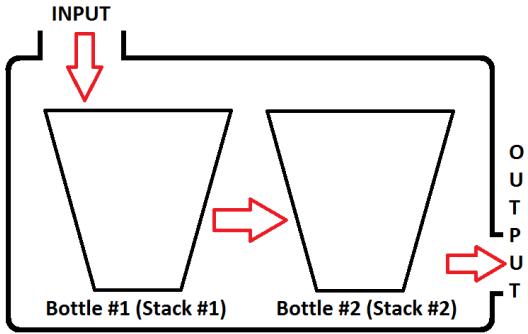
## B - Using Two Stacks As A Queue

On previous part, I've explained how can we reverse the order of stack elements. This was important, because if we push and pop elements to the stack, the output will be exactly in reverse order of a queue. Thinking on an example, let's push the array of integers

`{1, 2, 3, 4, 5}` to a stack. If we pop the elements and print them until the stack is empty, we will get the array in the reverse order of pushing order, which will be `{5, 4, 3, 2, 1}`

Remember that for the same input, if we dequeue the queue until the queue is empty, the output will be `{1, 2, 3, 4, 5}`. So it is obvious that for the same input order of elements, output of the queue is exactly reverse of the output of a stack. As we know how to reverse a stack using an extra stack, we can construct a queue using two stacks.

Our queue model will consist of two stacks. One stack will be used for `enqueue` operation (stack #1 on the left, will be called as Input Stack), another stack will be used for the `dequeue` operation (stack #2 on the right, will be called as Output Stack). Check out the image below;



Our pseudo-code is as below;

## Enqueue Operation

```
Push every input element to the Input Stack
```

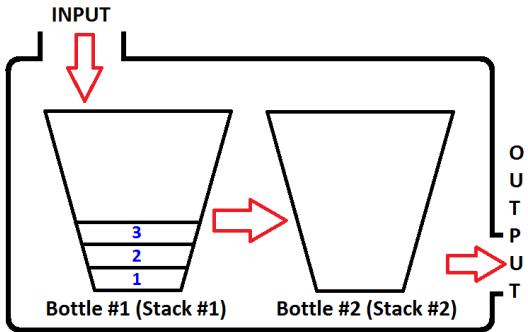
## Dequeue Operation

```

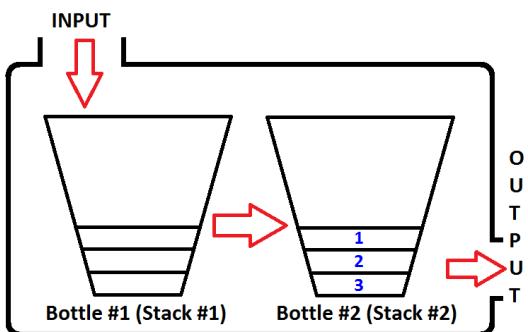
1 If (Output Stack is Empty)
2 pop every element in the Input Stack
3 and push them to the Output Stack until Input Stack is Empty
4
5 pop from Output Stack

```

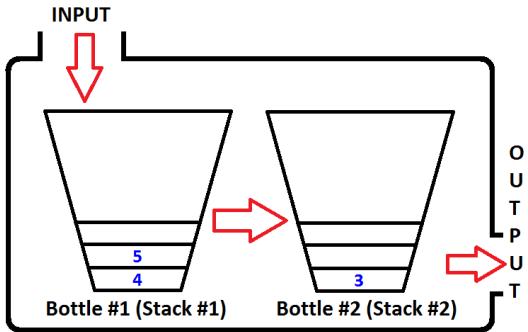
Let's enqueue the integers {1, 2, 3} respectively. Integers will be pushed on the **Input Stack (Stack #1)** which is located on the left;



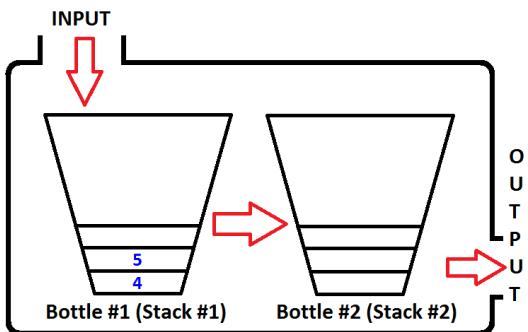
Then what will happen if we execute a dequeue operation? Whenever a dequeue operation is executed, queue is going to check if the Output Stack is empty or not(see the pseudo-code above) If the Output Stack is empty, then the Input Stack is going to be extracted on the output so the elements of Input Stack will be reversed. Before returning a value, the state of the queue will be as below;



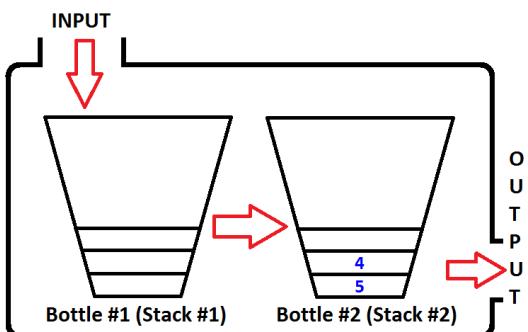
Check out the order of elements in the Output Stack (Stack #2). It's obvious that we can pop the elements from the Output Stack so that the output will be same as if we dequeued from a queue. Thus, if we execute two dequeue operations, first we will get {1, 2} respectively. Then element 3 will be the only element of the Output Stack, and the Input Stack will be empty. If we enqueue the elements 4 and 5, then the state of the queue will be as follows;



Now the Output Stack is not empty, and if we execute a dequeue operation, only 3 will be popped out from the Output Stack. Then the state will be seen as below;



Again, if we execute two more dequeue operations, on the first dequeue operation, queue will check if the Output Stack is empty, which is true. Then pop out the elements of the Input Stack and push them to the Output Stack until the Input Stack is empty, then the state of the Queue will be as below;



Easy to see, the output of the two dequeue operations will be {4, 5}

## **Wk-20-Notes**

# D1 Notes

## D1

---

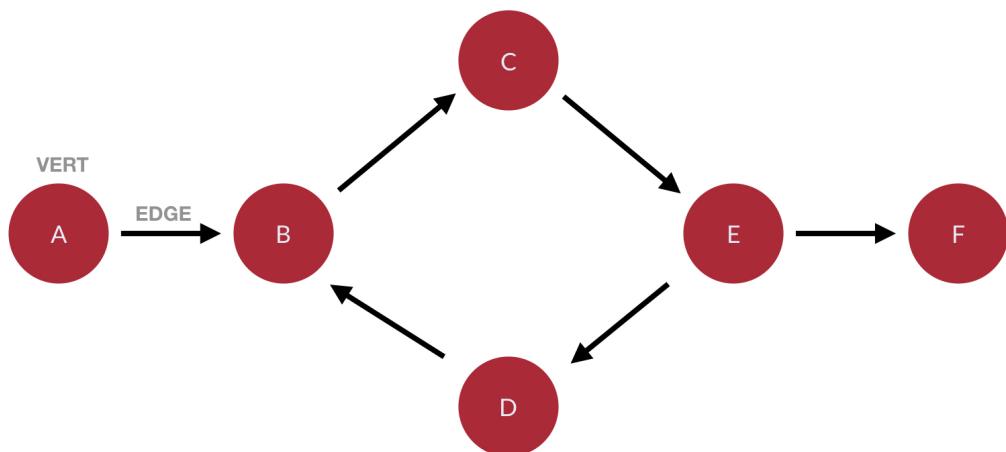
**Objective 01 - Describe what a graph is, explain its components, provide examples of its useful applications, and draw each of the different graph types**

### Overview

#### What Are Graphs?

Graphs are collections of related data. They're like trees, except connections can be made from any node to any other node, even forming loops. By this definition, *all trees are graphs, but not all graphs are trees.*

#### Components of Graphs



70733a2f2f692e696d6775722e636f6d2f456232536b68482e6a7067

We call the nodes in a graph **vertexes** (or **vertices** or **verts**), and we call the connections between the verts **edges**.

An edge denotes a relationship or linkage between the two verts.

## What Graphs Represent

Graphs can represent any multi-way relational data.

A graph could show a collection of cities and their linking roads.

It could show a collection of computers on a network.

It could show a population of people who know each other and Kevin Bacon (Links to an external site.).

It could represent trade relationships between nations.

It could represent the money owed in an ongoing poker night amongst friends.

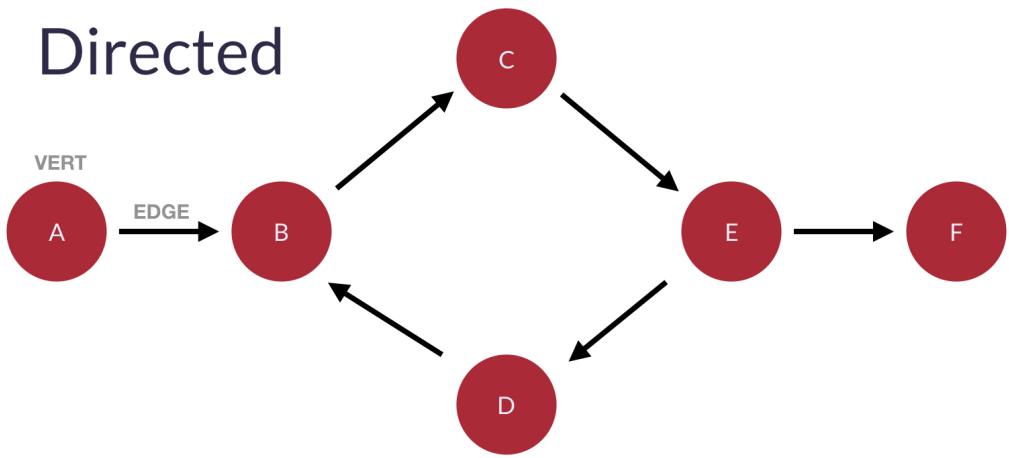
And so on.

## Types of Graphs

### Directed and Undirected Graphs

The nature of the relationship that we represent determines if we should use a directed or undirected graph. If we could describe the relationship as "one way", then a directed graph makes the most sense. For example, representing the owing of money to others (debt) with a directed graph would make sense.

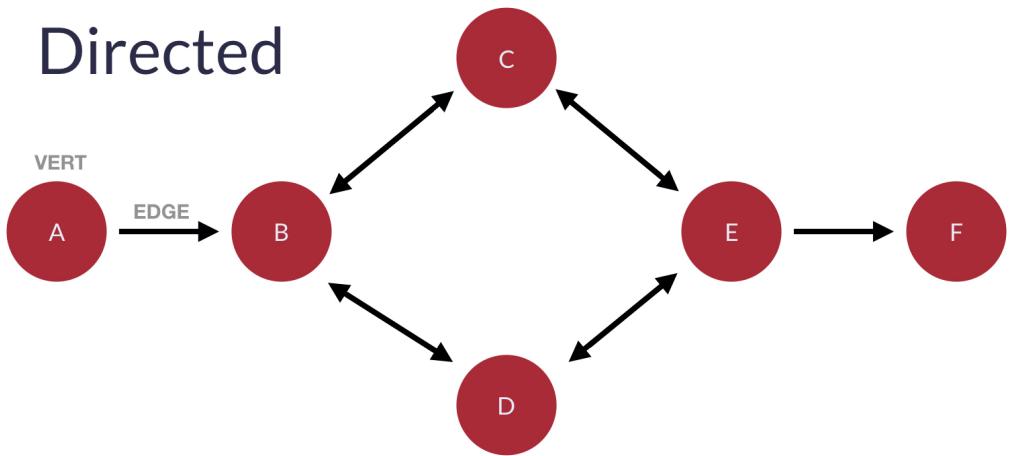
## Directed



<https://camo.githubusercontent.com/a17434989386f6f18a16851b5aeb8bbf92a129eb/68747470733a2f2f692e696d6775722e636f6d2f766677527244522e6a7067>

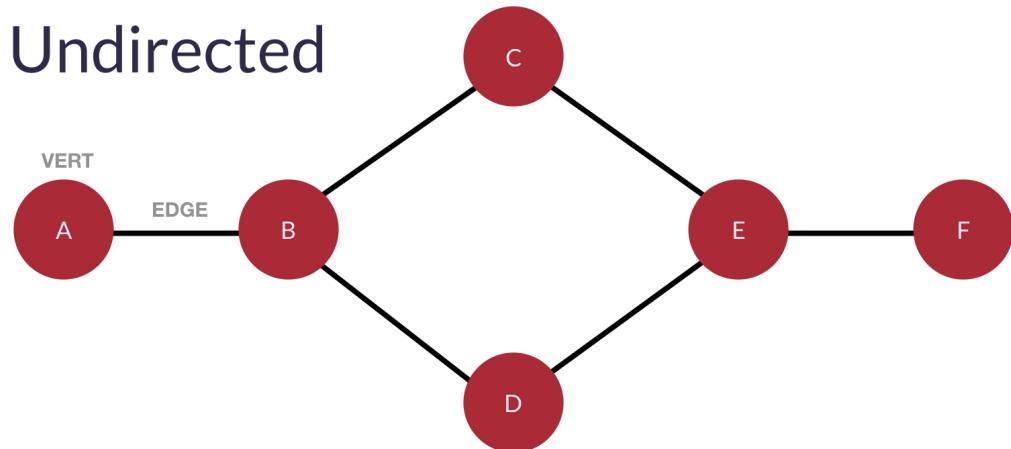
Directed graphs can also be bidirectional. For example, road maps are directed since all roads are one-way; however, most streets consist of lanes in both directions.

## Directed



<https://camo.githubusercontent.com/a6113174942b0d66c04a6e12bc67db2a8b8959d5/68747470733a2f2f692e696d6775722e636f6d2f6d386d4133676f2e6a7067>

If the relationship's nature is a mutual exchange, then an undirected graph makes the most sense. For example, we could use an undirected graph to represent users who have exchanged money in the past. Since an "exchange" relationship is always mutual, an **undirected** graph makes the most sense here.

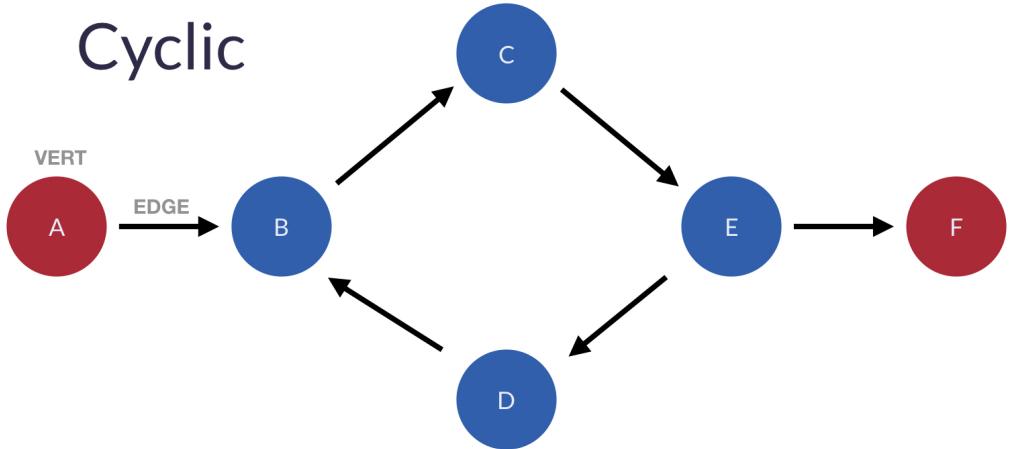


<https://camo.githubusercontent.com/6b782a86e1920d53411b88e1510c7efd0bed2bc2/68747470733a2f2f692e696d6775722e636f6d2f534a4e3036776a2e6a7067>

### Cyclic and Acyclic Graphs

If you can form a cycle (for example, follow the edges and arrive again at an already-visited vert), the graph is **cyclic**. For instance, in the image below, you can start at B and then follow the edges to C, E, D, and then back to B (which you've already visited).

## Cyclic

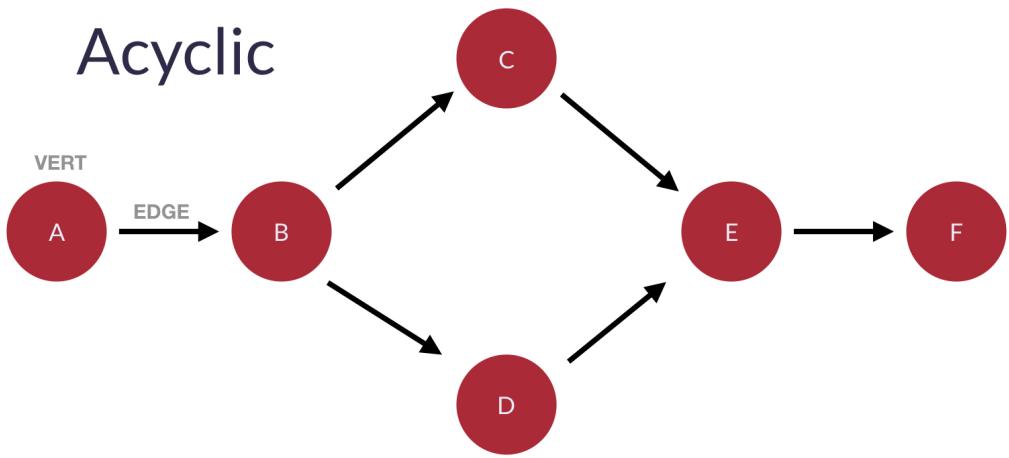


<https://camo.githubusercontent.com/e23529f1bd2dfe3227dee64fe174252b0c310d1a/68747470733a2f2f692e696d6775722e636f6d2f58764d44616c302e6a7067>

*Note: any undirected graph is automatically cyclic since you can always travel back across the same edge.*

If you cannot form a cycle (for example, you cannot arrive at an already-visited vert by following the edges), we call the graph **acyclic**. In the example below, no matter which vert you start at, you cannot follow edges in such a way that you can arrive at an already-visited vert.

## Acyclic

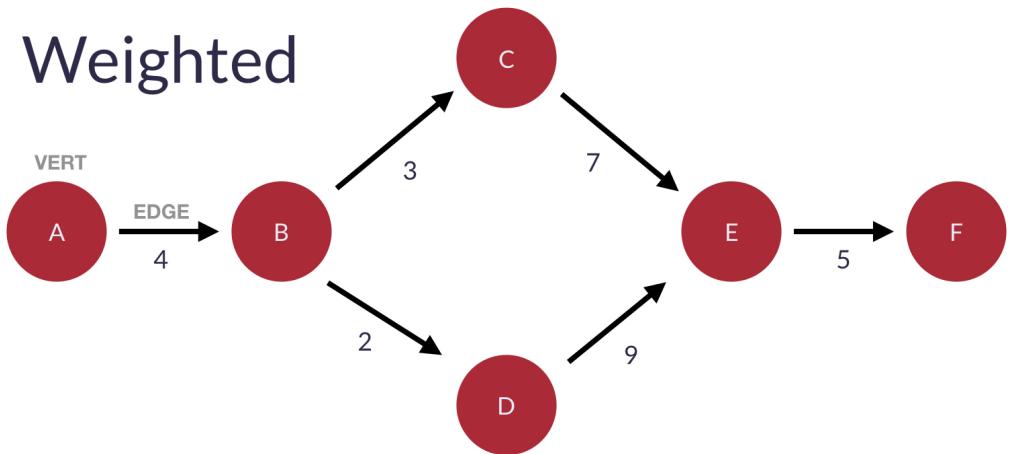


<https://camo.githubusercontent.com/321029108b001c2f2c3ea86c775f6a87b8436d6d/68747470733a2f2f692e696d6775722e636f6d2f4c58416d376d762e6a7067>

## Weighted Graphs

**Weighted graphs** have values associated with the edges. We call the specific values assigned to each edge **weights**.

## Weighted



<https://camo.githubusercontent.com/405834c88f7cd436bc69aebf9bc3f20072857d3e/68747470733a2f2f692e696d6775722e636f6d2f726a4d6a716b332e6a7067>

The weights represent different data in different graphs. In a graph representing road segments, the weights might represent the length of the road. The higher the total weight of a route on the graph, the longer the trip is. The weights can help decide which particular path we should choose when comparing all routes.

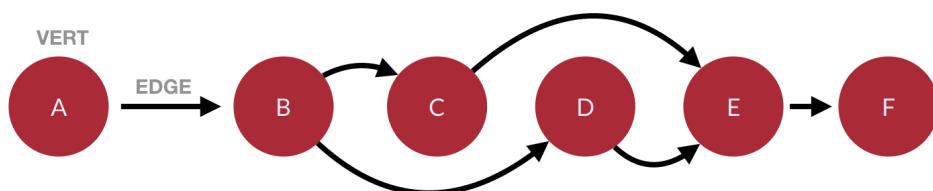
We can further modify weights. For example, if you were building a graph representing a map for bicycle routes, we could give roads with bad car traffic or very steep inclines unnaturally large weights. That way, a routing algorithm would be unlikely to take them. (This is how Google Maps avoids freeways when you ask it for walking directions.)

*Note: Dijkstra's Algorithm (Links to an external site.) is a graph search variant that accounts for edge weights.*

## Directed Acyclic Graphs (DAGs)

A **directed acyclic graph (DAG)** is a directed graph with no cycles. In other words, we can order a DAG's vertices linearly in such a way that every edge is directed from earlier to later in the sequence.

# Directed Acyclic



<https://camo.githubusercontent.com/4a276dc53718bd990dd5a8e62e99352f28965f0e/68747470733a2f2f692e696d6775722e636f6d2f6e71684d37757a2e6a7067>

A DAG has several applications. DAGs can model many different kinds of information. Below is a small list of possible applications:

- A spreadsheet where a vertex represents each cell and an edge for where one cell's formula uses another cell's value.
- The milestones and activities of large-scale projects where a topological ordering can help optimize the projects' schedule to use as little time as possible.
- Collections of events and their influence on each other like family trees or version histories.

It is also notable that git uses a DAG to represent commits. A commit can have a child commit, or more than one child commit (in a branch). A child could come from one parent commit or two (in the case of a merge). But there's no way to go back and form a repeating loop in the git commit hierarchy.

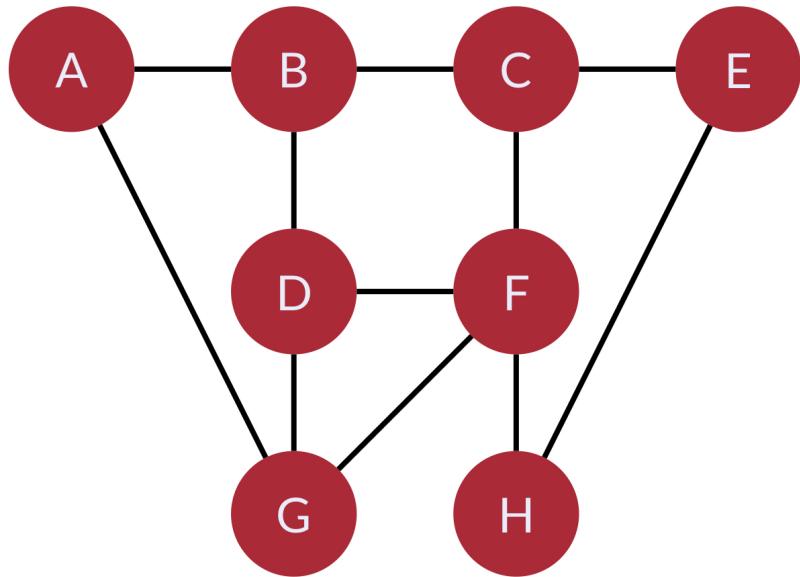
## Follow Along

Before you draw graphs on your own, let's draw some graphs together. For each graph, we will have a description.

### Exercise 1

*Draw an undirected graph of 8 verts.*

Remember, from our definitions above that an undirected graph has bidirectional edges. So, we can draw eight verts and then connect them with solid lines (not arrows) anyway we see fit.

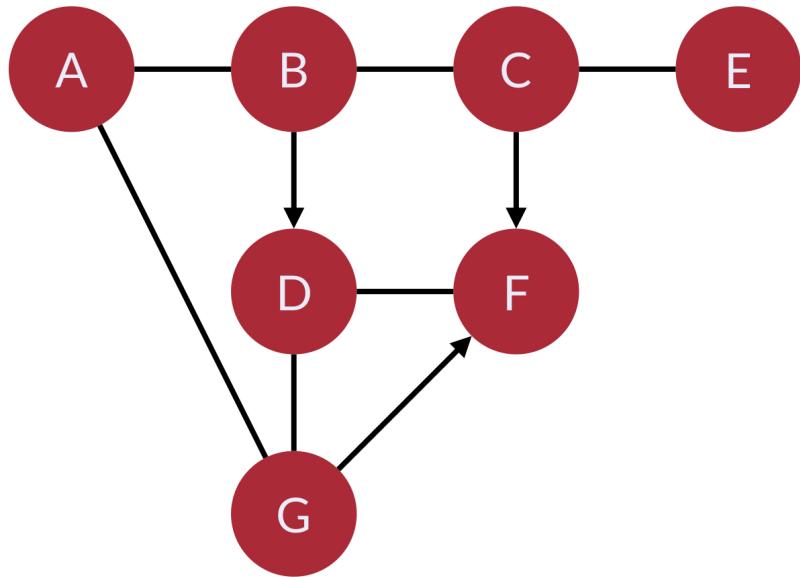


<https://camo.githubusercontent.com/de48eaaa53f7fead48f3d09e394f8b1342f7d226/68747470733a2f2f692e696d6775722e636f6d2f6d6964443358642e6a7067>

## Exercise 2

*Draw a directed graph of 7 verts.*

A directed graph has at least one edge that is *not* bidirectional. So, again, we can draw our seven verts and then connect them with edges. This time, we need to make sure that one of the edges is an arrow pointing in only one direction.



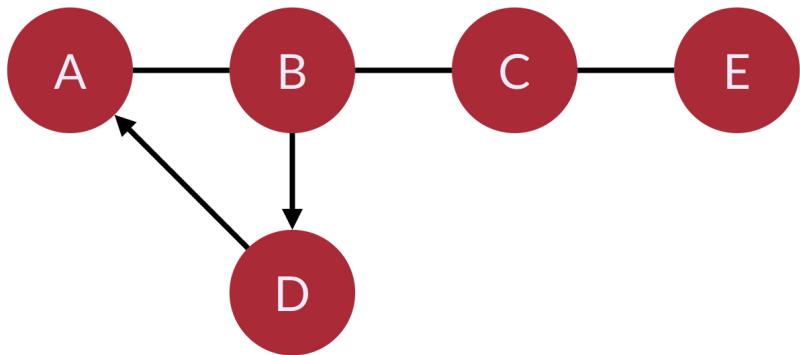
<https://camo.githubusercontent.com/53c2b80679e6731818f5bb72fecdf17579dd0f70/68747470733a2f2f692e696d6775722e636f6d2f3870436f6568412e6a7067>

### Exercise 3

*Draw a cyclic directed graph of 5 verts.*

This drawing will be similar to one for Exercise 2 because it is a directed graph. However, in this graph, we also need to ensure that it has at least one cycle. Remember that a cycle is when you can follow the graph's edges and arrive at a vertex that you've already visited.

To draw this graph, we will draw our five verts and then draw our edges, making sure that we create at least one cycle.



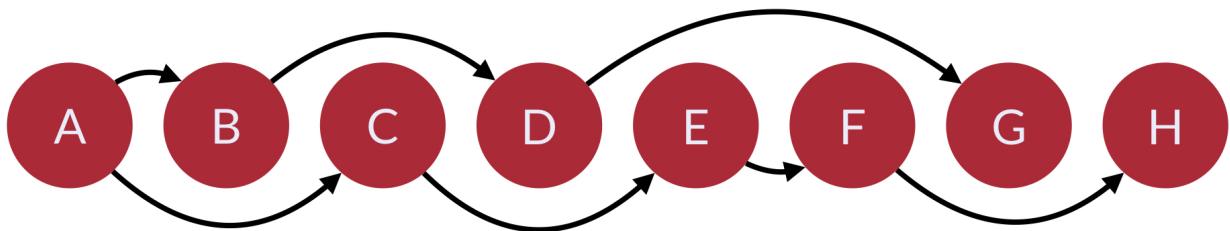
<https://camo.githubusercontent.com/8609287c7507ded66414ab2be7154d68436b82ac/68747470733a2f2f692e696d6775722e636f6d2f4a424f7572506e2e6a7067>

#### Exercise 4

*Draw a directed acyclic graph (DAG) of 8 verts.*

Again, this graph will be directed. The difference is that it will be acyclic—we can order a DAG's vertices linearly so that every edge is directed from earlier to later in the sequence.

For this graph, we will draw our eight verts in a line from left to right. We will then draw our edges, making sure that the edges always point from left to right (earlier to later in the sequence).



<https://camo.githubusercontent.com/f9f74a9565045797142f292f619840371fc04698/68747470733a2f2f692e696d6775722e636f6d2f4d4e4c5a6f6f482e6a7067>

## Challenge

Draw one graph for each of the descriptions below:

1. Undirected graph of 4 verts.
2. Directed graph of 5 verts.
3. Cyclic directed graph of 6 verts.
4. DAG of 7 verts.

## Additional Resources

- <https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8>

---

**Objective 02 - Represent a graph as an adjacency list and an adjacency matrix and compare and contrast the respective representations**

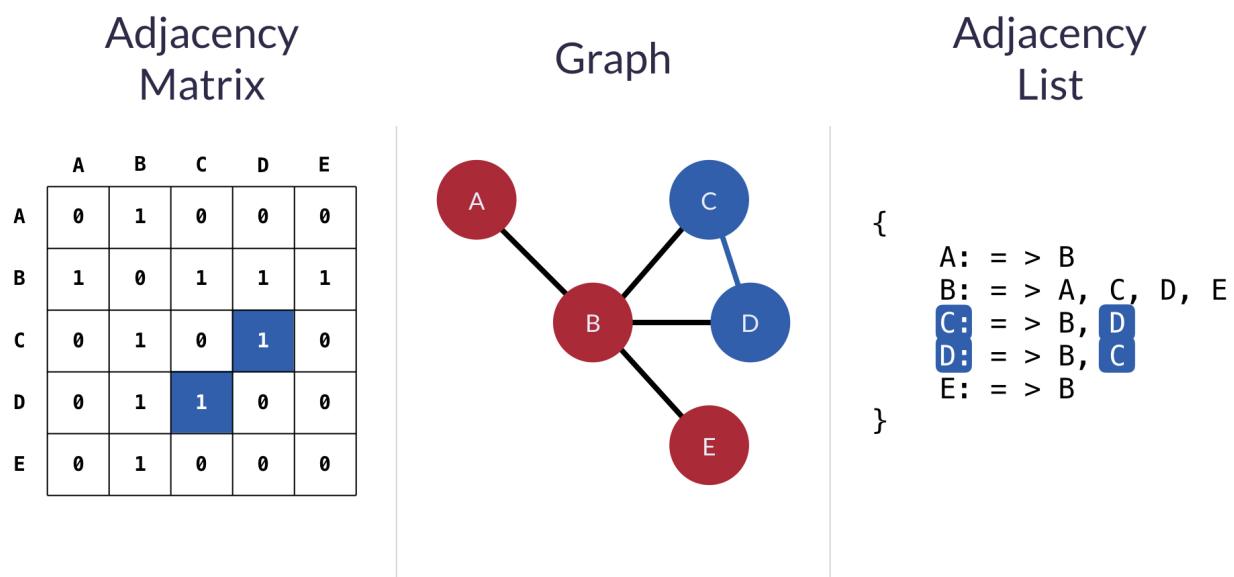
# Overview

## Graph Representations

Two common ways to represent graphs in code are **adjacency lists** and **adjacency matrices**.

Both of these options have strengths and weaknesses. When deciding on a graph implementation, it's essential to understand what type of data you will store and what operations you need to run on the graph.

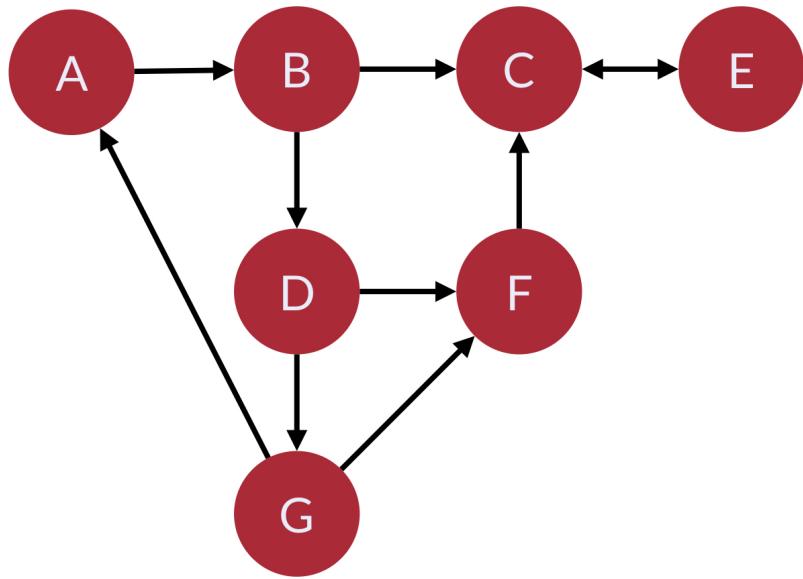
Below is an example of how we would represent a graph with an adjacency matrix and an adjacency list. Notice how we represent the relationship between verts C and D when using each type.



<https://camo.githubusercontent.com/ff694105bfdaea68ee3a73c75cf604ac8f020e1c/68747470733a2f2f692e696d6775722e636f6d2f7369476d7138582e6a7067>

## Adjacency List

In an adjacency list, the graph stores a list of vertices. For each vertex, it holds a list of each connected vertex.



<https://camo.githubusercontent.com/0e81024228bd0b1dd29f33c47b0896b7a978e911/68747470733a2f2f692e696d6775722e636f6d2f476953746d4e682e6a7067>

Below is a representation of the graph above in Python:

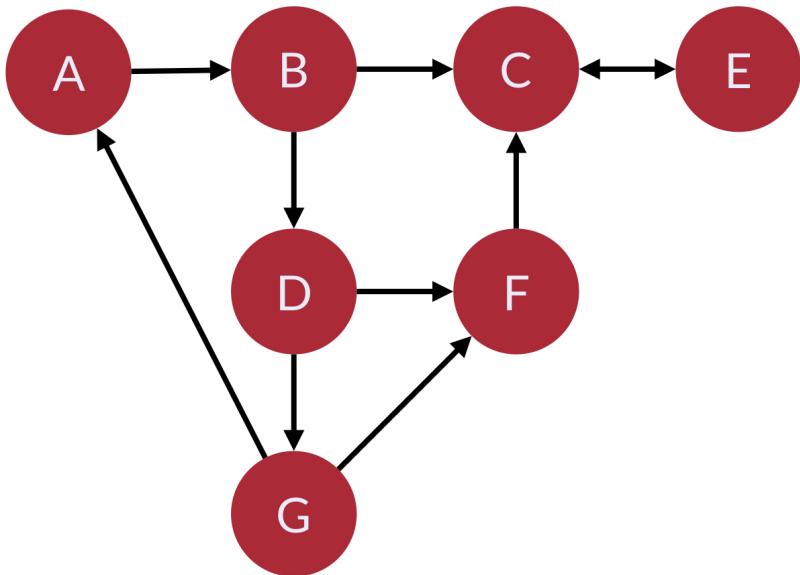
```

1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B"},
5 "B": {"C", "D"},
6 "C": {"E"},
7 "D": {"F", "G"},
8 "E": {"C"},
9 "F": {"C"},
10 "G": {"A", "F"}
11 }

```

Notice that this adjacency *list* doesn't use any lists. The `vertices` collection is a `dictionary` which lets us access each collection of edges in  $O(1)$  constant time. Because a `set` contains the edges, we can check for edges in  $O(1)$  constant time.

## Adjacency Matrix



<https://camo.githubusercontent.com/0e81024228bd0b1dd29f33c47b0896b7a978e911/68747470733a2f2f692e696d6775722e636f6d2f476953746d4e682e6a7067>

Here is the representation of the graph above in an adjacency matrix:

```

1 class Graph:
2 def __init__(self):
3 self.edges = [[0,1,0,0,0,0,0],
4 [0,0,1,1,0,0,0],
5 [0,0,0,0,1,0,0],
6 [0,0,0,0,0,1,1],
7 [0,0,1,0,0,0,0],
8 [0,0,1,0,0,0,0],
9 [1,0,0,0,0,1,0]]

```

We represent this matrix as a two-dimensional array—a list of lists. With this implementation, we get the benefit of built-in edge weights. `0` denotes no relationship, but any other value represents an edge label or edge weight. The drawback is that we do not have a built-in association between the vertex values and their index.

In practice, implementing both the adjacency list and adjacency matrix would contain more information by including `Vertex` and `Edge` classes.

## Tradeoffs

Adjacency matrices and adjacency lists have strengths and weaknesses. Let's explore their tradeoffs by comparing their attributes and the efficiency of operations.

In all the following examples, we are using the following shorthand to denote the graph's properties:

Shorthand	Property
V	Total number of vertices in the graph
E	Total number of edges in the graph
e	Average number of edges per vertex

## Space Complexity

### Adjacency Matrix

*Complexity:  $O(V^2)$  space*

Consider a dense graph where each vertex points to each other vertex. Here, the total number of edges will approach  $V^2$ . This fact means that regardless of whether you choose an adjacency list or an adjacency matrix, both will have a comparable space complexity. However, dictionaries and sets are less space-efficient than lists. So, for dense graphs (graphs with a high average number of edges per vertex), the adjacency matrix is more efficient because it uses lists instead of dictionaries and sets.

### Adjacency List

*Complexity:  $O(V+E)$  space*

Consider a sparse graph with 100 vertices and only one edge. An adjacency list would have to store all 100 vertices but only needs to keep track of that single edge. The adjacency matrix would need to store  $100 \times 100 = 10,000$  connections, even though all but one would be 0.

*Takeaway: The worst-case storage of an adjacency list occurs when the graph is dense. The matrix and list representation have the same complexity ( $O(V^2)$ ). However, for the general case, the list representation is usually more desirable. Also, since finding a vertex's neighbors is*

a common task, and adjacency lists make this operation more straightforward, it is most common to use adjacency lists to represent graphs.

## Add Vertex

### Adjacency Matrix

Complexity:  $O(V)$  time

For an adjacency matrix, we would need to add a new value to the end of each existing row and add a new row.

```
1 for v in self.edges:
2 self.edges[v].append(0)
3 v.append([0] * len(self.edges) + 1)
```

for v in self.edges: self.edges[v].append(0) v.append([0] \* len(self.edges) + 1))

Remember that with Python lists, appending to the end of a list is  $O(1)$  because of over-allocation of memory but can be  $O(n)$  when the over-allocated memory fills up. When this occurs, adding the vertex can be  $O(V^2)$ .

### Adjacency List

Complexity:  $O(1)$  time

Adding a vertex is simple in an adjacency list:

```
self.vertices["H"] = set()
```

Adding a new key to a dictionary is a constant-time operation.

*Takeaway: Adding vertices is very inefficient for adjacency matrices but very efficient for adjacency lists.*

## Remove Vertex

### Adjacency Matrix

Complexity:  $O(V^2)$

Removing vertices is inefficient in both representations. In an adjacency matrix, we need to remove the removed vertex's row and then remove that column from each row. Removing an element from a list requires moving everything after that element over by one slot, which takes an average of  $V/2$  operations. Since we need to do that for every single row in our matrix, that results in  $V^2$  time complexity. We need to reduce each vertex index after our removed index by one as well, which doesn't add to our quadratic time complexity but adds extra operations.

### Adjacency List

Complexity:  $O(V)$

We need to visit each vertex for an adjacency list and remove all edges pointing to our removed vertex. Removing elements from sets and dictionaries is an  $O(1)$  operation, resulting in an overall  $O(V)$  time complexity.

*Takeaway: Removing vertices is inefficient in both adjacency matrices and lists but more efficient in lists.*

## Add Edge

### Adjacency Matrix

Complexity:  $O(1)$

Adding an edge in an adjacency matrix is simple:

```
self.edges[v1][v2] = 1
```

### Adjacency List

Complexity:  $O(1)$

Adding an edge in an adjacency list is simple:

```
self.vertices[v1].add(v2)
```

Both are constant-time operations.

*Takeaway: Adding edges to both adjacency matrices and lists is very efficient.*

## Remove Edge

### Adjacency Matrix

*Complexity: O(1)*

Removing an edge from an adjacency matrix is simple:

```
self.edges[v1][v2] = 0
```

### Adjacency List

*Complexity: O(1)*

Removing an edge from an adjacency list is simple:

```
self.vertices[v1].remove(v2)
```

Both are constant-time operations.

*Takeaway: Removing edges from both adjacency matrices and lists is very efficient.*

## Find Edge

### Adjacency Matrix

*Complexity:*  $O(1)$

Finding an edge in an adjacency matrix is simple:

```
return self.edges[v1][v2] > 0
```

## Adjacency List

*Complexity:*  $O(1)$

Finding an edge in an adjacency list is simple:

```
return v2 in self.vertices[v1]
```

Both are constant-time operations.

*Takeaway:* Finding edges in both adjacency matrices and lists is very efficient.

## Get All Edges from Vertex

You can use several commands if you want to know all the edges originating from a particular vertex.

## Adjacency Matrix

*Complexity:*  $O(V)$

In an adjacency matrix, this is complicated. You would need to iterate through the entire row and populate a list based on the results:

```
1 v_edges = []
2 for v2 in self.edges[v]:
3 if self.edges[v][v2] > 0:
4 v_edges.append(v2)
5 return v_edges
```

## Adjacency List

*Complexity:*  $O(1)$

With an adjacency list, this is as simple as returning the value from the vertex dictionary:

```
return self.vertex[v]
```

*Takeaway:* Fetching all edges is less efficient in an adjacency matrix than an adjacency list.

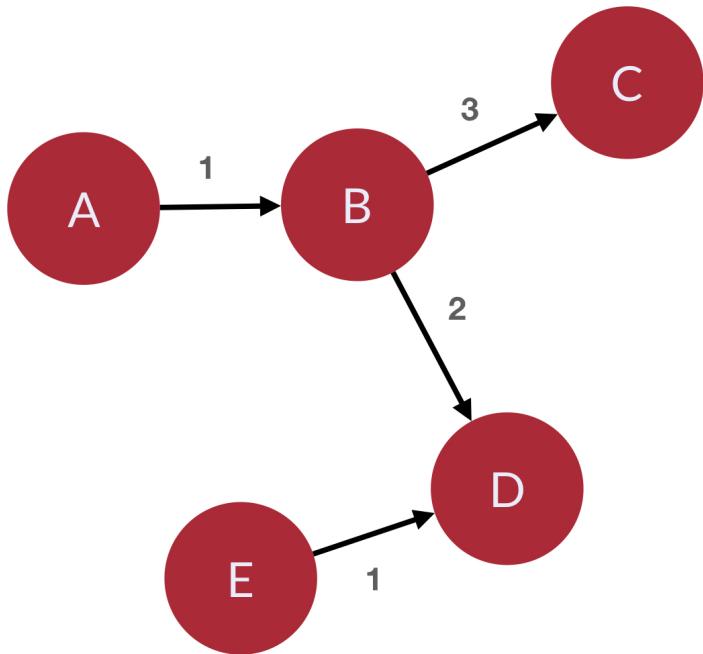
## Summary

Let's summarize all this complexity information in a table:

type	Space	Add Vert	Remove Vert	Add Edge	Remove Edge	Find Edge	Get All Edges
Matrix	$O(V^2)$	$O(V)$	$O(V^2)$	$O(1)$	$O(1)$	$O(1)$	$O(V)$
List	$O(V+E)$	$O(1)$	$O(V)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

In most practical use-cases, an adjacency list will be a better choice for representing a graph. However, it is also crucial that you be familiar with the matrix representation. Why? Because there are some dense graphs or weighted graphs that could have better space efficiency when represented by a matrix.

## Follow Along



<https://camo.githubusercontent.com/335012587396b095af8f6a8f28e2d2aedb3d84d0/68747470733a2f2f692e696d6775722e636f6d2f796931503441462e6a7067>

Together, we will now use the graph shown in the picture above and represent it in both an adjacency list and an adjacency matrix.

### Adjacency List

First, the adjacency list:

```

1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B": 1},
5 "B": {"C": 3, "D": 2},
6 "C": {},
7 "D": {},
8 "E": {"D": 1}
9 }
10
```

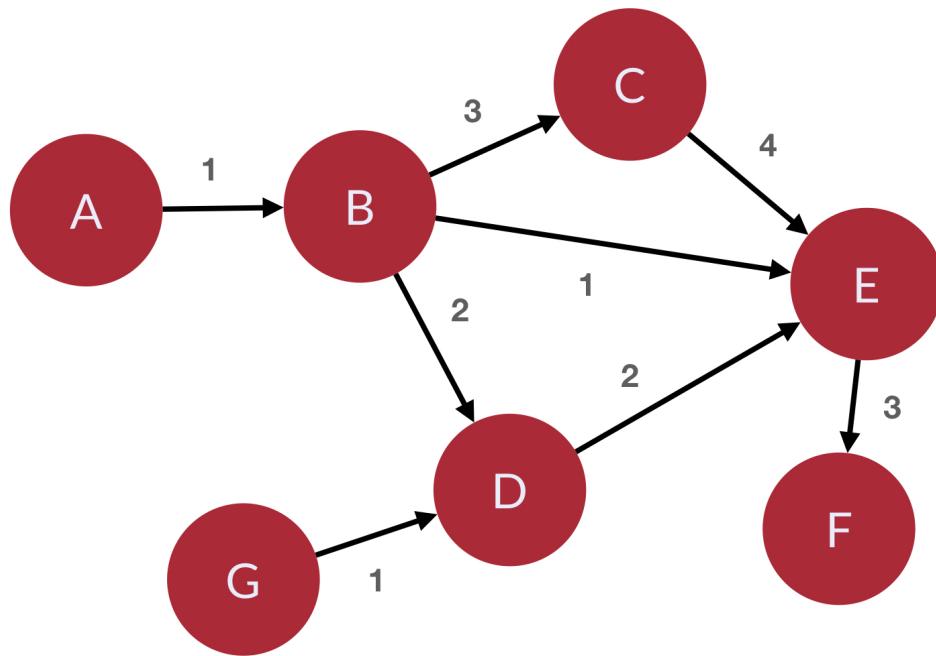
The difference between this implementation and the previous adjacency list is that this representation allows our edges to have weights.

## Adjacency Matrix

Now, we need to implement an adjacency matrix. Remember, that one benefit of the matrix is how easy it is to represent edge weights:

```
1 class Graph:
2 def __init__(self):
3 self.edges = [[0,1,0,0,0],
4 [0,0,3,2,0],
5 [0,0,0,0,0],
6 [0,0,0,0,0],
7 [0,0,0,1,0]]
```

## Challenge



<https://camo.githubusercontent.com/b6251eb484344b565ae2753682c645f85283ab28/68747470733a2f2f692e696d6775722e636f6d2f634a366c656b4d2e6a7067>

1. Using the graph shown in the picture above, write python code to represent the graph in an adjacency list.
2. Using the same graph you used for the first exercise, write python code to represent the graph in an adjacency matrix.

3. Write a paragraph that compares and contrasts the efficiency of your different representations.

## Additional Resources

- <https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs>
- 

# Objective 03 - Implement user-defined Vertex and Graph classes that allow basic operations

## Overview

We will now use dictionaries to implement the graph abstract data type in Python. We need to have two classes. First, the `Graph` class that will keep track of the vertices in the graph instance. Second, the `Vertex` class, which we will use to represent each vertex contained in a graph. Both classes will have methods that allow you to complete the basic operations for interacting with graphs and vertices.

## Follow Along

### The `Vertex` Class

Let's start by defining a `Vertex` class and defining its initialization method (`__init__`) and its `__str__` method so we can print out a human-readable string representations of each vertex:

```
1 class Vertex:
2 def __init__(self, value):
3 self.value = value
4 self.connections = {}
5
6 def __str__(self):
7 return str(self.value) + ' connections: ' + str([x.value for x in sel
```

The next thing we need for our `Vertex` class is a way to other vertices that are connected and the `weight` of the connection edge. We will call this method `add_connection`.

```
1 class Vertex:
2 def __init__(self, value):
3 self.value = value
4 self.connections = {}
5
6 def __str__(self):
7 return str(self.value) + ' connections: ' + str([x.value for x in self.connections])
8
9 def add_connection(self, vert, weight = 0):
10 self.connections[vert] = weight
```

Let's now add three methods that allow us to get data out of our `Vertex` instance objects.

These three methods will be `get_connections` (retrieves all currently connected vertices), `get_value` (retrieves the value of the vertex instance), and `get_weight` (gets the edge weight from the vertex to a specified connected vertex).

```
1 class Vertex:
2 def __init__(self, value):
3 self.value = value
4 self.connections = {}
5
6 def __str__(self):
7 return str(self.value) + ' connections: ' + str([x.value for x in self.connections])
8
9 def add_connection(self, vert, weight = 0):
10 self.connections[vert] = weight
11
12 def get_connections(self):
13 return self.connections.keys()
14
15 def get_value(self):
16 return self.value
17
18 def get_weight(self, vert):
19 return self.connections[vert]
```

We've finished our `Vertex` class. Now, let's work on our `Graph` class.

## The Graph Class

Our graph class's primary purpose is to be a way that we can map vertex names to specific vertex objects. We also want to keep track of the number of vertices that our graph contains using a `count` property. We will do so using a dictionary. Let's start by defining an initialization method (`__init__`).

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
```

Next, we need a way to add vertices to our graph. Let's define an `add_vertex` method.

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
5
6 def add_vertex(self, value):
7 self.count += 1
8 new_vert = Vertex(value)
9 self.vertices[value] = new_vert
10 return new_vert
```

We also need a way to add an edge to our graph. We need a method that can create a connection between two vertices and specify the edge's weight. Let's do so by defined an `add_edge` method.

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
5
6 def add_vertex(self, value):
7 self.count += 1
8 new_vert = Vertex(value)
9 self.vertices[value] = new_vert
10 return new_vert
```

```
11
12 def add_edge(self, v1, v2, weight = 0):
13 if v1 not in self.vertices:
14 self.add_vertex(v1)
15 if v2 not in self.vertices:
16 self.add_vertex(v2)
17 self.vertices[v1].add_connection(self.vertices[v2], weight)
```

Next, we need a way to retrieve a list of all the vertices in our graph. We will define a method called `get_vertices`.

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
5
6 def add_vertex(self, value):
7 self.count += 1
8 new_vert = Vertex(value)
9 self.vertices[value] = new_vert
10 return new_vert
11
12 def add_edge(self, v1, v2, weight = 0):
13 if v1 not in self.vertices:
14 self.add_vertex(v1)
15 if v2 not in self.vertices:
16 self.add_vertex(v2)
17 self.vertices[v1].add_connection(self.vertices[v2], weight)
18
19 def get_vertices(self):
20 return self.vertices.keys()
```

Last, we will override a few built-in methods (`__contains__` and `__iter__`) that are available on objects to make sure they work correctly with `Graph` instance objects.

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
5
6 def __contains__(self, vert):
7 return vert in self.vertices
8
```

```

9 def __iter__(self):
10 return iter(self.vertices.values())
11
12 def add_vertex(self, value):
13 self.count += 1
14 new_vert = Vertex(value)
15 self.vertices[value] = new_vert
16 return new_vert
17
18 def add_edge(self, v1, v2, weight = 0):
19 if v1 not in self.vertices:
20 self.add_vertex(v1)
21 if v2 not in self.vertices:
22 self.add_vertex(v2)
23 self.vertices[v1].add_connection(self.vertices[v2], weight)
24
25 def get_vertices(self):
26 return self.vertices.keys()

```

Let's go ahead and test our class definitions and build up a graph structure in a Python interactive environment.

```

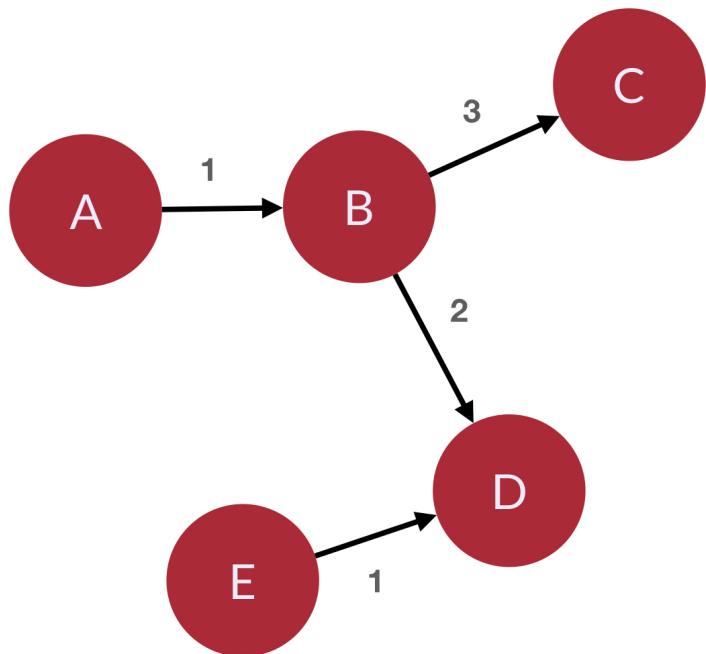
1 >>> g = Graph()
2 >>> for i in range(8):
3 ... g.add_vertex(i)
4 ...
5 <__main__.Vertex object at 0x7fd0f183f5e0>
6 <__main__.Vertex object at 0x7fd0f183fdc0>
7 <__main__.Vertex object at 0x7fd0f183fe20>
8 <__main__.Vertex object at 0x7fd0f183fb50>
9 <__main__.Vertex object at 0x7fd0f183fee0>
10 <__main__.Vertex object at 0x7fd0f183ff40>
11 <__main__.Vertex object at 0x7fd0f183ffd0>
12 <__main__.Vertex object at 0x7fd0f183ffa0>
13 >>> g.vertices
14 {0: <__main__.Vertex object at 0x7fd0f183f5e0>, 1: <__main__.Vertex object at
15 >>> g.add_edge(0,1,3)
16 >>> g.add_edge(0,7,2)
17 >>> g.add_edge(1,3,4)
18 >>> g.add_edge(2,2,1)
19 >>> g.add_edge(3,6,5)
20 >>> g.add_edge(4,0,2)
21 >>> g.add_edge(5,2,3)
22 >>> g.add_edge(5,3,1)
23 >>> g.add_edge(6,2,3)
24 >>> g.add_edge(7,1,4)
25 >>> for v in g:
26 ... for w in v.get_connections():

```

```
27 ... print("(%s, %s)" % (v.get_value(), w.get_value()))
28 ...
29 (0, 1)
30 (0, 7)
31 (1, 3)
32 (2, 2)
33 (3, 6)
34 (4, 0)
35 (5, 2)
36 (5, 3)
37 (6, 2)
38 (7, 1)
39 >>>
```

## Challenge

Load the `Vertex` class and `Graph` class into an interactive Python environment and use the classes to create an instance of the graph shown below.



<https://camo.githubusercontent.com/335012587396b095af8f6a8f28e2d2aedb3d84d0/68747470733a2f2f692e696d6775722e636f6d2f796931503441462e6a7067>

## Additional Resources

- <https://www.geeksforgeeks.org/generate-graph-using-dictionary-python/> (Links to an external site.)

## D3

### What is a Hash table or a Hashmap in Python?

In computer science, a Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). It basically makes use of a function that computes an index value that in turn holds the elements to be searched, inserted, removed, etc. This makes it easy and fast to access data. In general, hash tables store key-value pairs and the key is generated using a hash function.

Hash tables or has maps in Python are implemented through the built-in dictionary data type. The keys of a dictionary in Python are generated by a hashing function. The elements of a dictionary are not ordered and they can be changed.

An example of a dictionary can be a mapping of employee names and their employee IDs or the names of students along with their student IDs.

Moving ahead, let's see the difference between the hash table and hashmap in Python.

---

### Hash Table vs hashmap: Difference between Hash Table and Hashmap in Python

Hash Table	Hashmap
Synchronized	Non-Synchronized
Fast	Slow
Allows one null key and more than one null values	Does not allows null keys or values

---

### Creating Dictionaries:

Dictionaries can be created in two ways:

- Using curly braces ({} )
- Using the *dict()* function

## Using curly braces:

Dictionaries in Python can be created using curly braces as follows:

### EXAMPLE:

```
123 my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe': '003'}print(my_dict)type(my_dict)
```

---

### OUTPUT:

```
{'Dave': '001', 'Ava': '002', 'Joe': '003'}
dict
```

## Using *dict()* function:

Python has a built-in function, *dict()* that can be used to create dictionaries in Python. For example:

### EXAMPLE:

```
123 new_dict=dict()print(new_dict)type(new_dict)
```

---

### OUTPUT:

```
{ }
dict
```

In the above example, an empty dictionary is created since no key-value pairs are supplied as a parameter to the *dict()* function. In case you want to add values, you can do as follows:

### EXAMPLE:

```
[REDACTED]
```

```
new_dict=dict(Dave = '001' , Ava= '002' , Joe='003')print(new_dict)type(new_dict)
```

---

## OUTPUT:

```
{'Dave': '001', 'Ava': '002', 'Joe': '003'}
dict
```

---

## Creating Nested Dictionaries:

Nested dictionaries are basically dictionaries that lie within other dictionaries. For example:

## EXAMPLE:

```
emp_details = {'Employee': {'Dave': {'ID': '001', 'Salary': 2000, 'Designation':'Python Developer'}, 'Ava': {'ID':'002', 'Salary': 2300, 'Designation': 'Java Developer'}, 'Joe': {'ID': '003', 'Salary': 1843, 'Designation': 'Hadoop Developer'}}}
```

---

## Performing Operations on Hash tables using Dictionaries:

There are a number of operations that can be performed on has tables in Python through dictionaries such as:

- Accessing Values
- Updating Values
- Deleting Element

## Accessing Values:

The values of a dictionary can be accessed in many ways such as:

- Using key values

- Using functions
- Implementing the for loop

### Using key values:

Dictionary values can be accessed using the key values as follows:

#### EXAMPLE:

12

```
my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe':
'003'}my_dict['Dave']
```

---

**OUTPUT:** '001'

### Using functions:

There are a number of built-in functions that can be used such as get(), keys(), values(), etc.

#### EXAMPLE:

```
my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe':
'003'}print(my_dict.keys())print(my_dict.values())print(my_dict.get('Dave'))
```

---

**OUTPUT:**

```
dict_keys(['Dave', 'Ava', 'Joe'])
dict_values(['001', '002', '003'])
001
```

### Implementing the for loop:

The for loop allows you to access the key-value pairs of a dictionary easily by iterating over them. For example:



```
my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe': '003'}print("All keys")for x in my_dict: print(x)
#prints the keysprint("All values")for x in my_dict.values(): print(x)
#prints valuesprint("All keys and values")for x,y in my_dict.items(): print(x, ":" , y) #prints keys and values
```

---

## OUTPUT:

```
All keys
Dave
Ava
Joe
All values
001
002
003
All keys and values
Dave : 001
Ava : 002
Joe : 003
```

## Updating Values:

Dictionaries are mutable data types and therefore, you can update them as and when required. For example, if I want to change the ID of the employee named Dave from '001' to '004' and if I want to add another key-value pair to my dictionary, I can do as follows:

## EXAMPLE:

```
my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe': '003'}my_dict['Dave'] = '004' #Updating the value of Davemy_dict['Chris'] = '005'
#adding a key-value pairprint(my_dict)
```

---

**OUTPUT:** {'Dave': '004', 'Ava': '002', 'Joe': '003', 'Chris': '005'}

## Deleting items from a dictionary:

There are a number of functions that allow you to delete items from a dictionary such as `del()`, `pop()`, `popitem()`, `clear()`, etc. For example:

**EXAMPLE:**

```
my_dict={'Dave': '004', 'Ava': '002', 'Joe': '003', 'Chris': '005'}del
my_dict['Dave'] #removes key-value pair of 'Dave'my_dict.pop('Ava')
#removes the value of 'Ava'my_dict.popitem()
#removes the last inserted itemprint(my_dict)
```

---

**OUTPUT:** {'Joe': '003'}

The above output shows that all the elements except 'Joe: 003' have been removed from the dictionary using the various functions.

**Converting Dictionary into a dataframe:**

As you have seen previously, I have created a nested dictionary containing employee names and their details mapped to it. Now to make a clear table out of that, I will make use of the pandas library in order to put everything as a dataframe.

**EXAMPLE:**

```
import pandas as pdemp_details = {'Employee': {'Dave': {'ID':
'001', 'Salary':
2000, 'Designation': 'Python Developer'}, 'Ava': {'ID': '002', 'Salary':
2300, 'Designation': 'Java Developer'}, 'Joe': {'ID': '003', 'Salary':
1843, 'Designation':
'Hadoop Developer'}}}df=pd.DataFrame(emp_details['Employee'])print(df)
```

---

**OUTPUT:**

		Dave	Ava	Joe
Designation	Python Developer	Java Developer	Hadoop Developer	
ID	001	002	003	
Salary	2000	2300	1843	

# Sprint Prep

**dev-enviorment-setup**

# Install Python

## Installing Python 3

Brian "Beej Jorgensen" Hall edited this page on May 29, 2020 · 1 revision

**NOTE: pipenv is *optional!* We don't use it in CS.** But it's a neat package manager if you get into more complex Python projects. It can be a headache of an install for some people. You can safely ignore anything about pipenv below if you don't want to mess with it.

We'll want to install Python 3 (version 3.x), which is the runtime for the language itself. The runtime is what allows you to execute Python code and files by typing

`python [file_or_code_to_execute]` in your terminal. You can also open up a Python REPL (Read-Eval-Print Loop) to quickly and easily mess around with Python code once you've gotten the runtime installed. If you recall how Node let's you execute and run JavaScript code locally on your machine instead of having to rely on the browser, the Python runtime pretty much let's you do the same thing but with Python code.

Additionally, we'll be talking about how to install the (optional) pipenv virtual environment manager. Think of it as the `npm` of Python (though pipenv is also capable of performing a bunch of other tasks as well, most notably running your Python projects in an isolated virtual environment).

### Note for Anaconda users

Unfortunately, we haven't found a way to get Anaconda to play nicely with pipenv. If you get them working together, please let your instructor know how you did it.

### Testing the Install

If you can run `python` or `python3` and see a 3.7 or later version, you're good to go:

```
1 $ python3 --version
2 Python 3.6.5
```

or on some systems, Python 3 is just `python` :

```
1 $ python --version
2 Python 3.6.5
```

And optionally try `pipenv` :

```
1 $ pipenv --version
2 pipenv, version [some remotely recent date, probably]
```

Otherwise, keep reading. :)

## macOS

While macOS comes with Python 2, we need to get Python 3 in there as well.

If you don't have Brew installed, follow the instructions on the brew website.

Use Brew to install Python and pipenv at the Terminal prompt:

```
brew install python pipenv
```

## Windows

**Note:** Git Bash doesn't seem to cooperate if you're trying to install Python on Windows. Try another terminal like Powershell.

Recommend updating Windows to the latest version.

### Windows Store

Python 3 is in the Windows Store and can be installed from there.

## Official Binaries

When installing the official package, be sure to check the

[ ] Add to PATH

checkbox!!

## Official Package

### Pipenv

This is what worked for Beej. YMMV.

1. Install Python, as per above.
2. Bring up a shell (cmd.exe or Powershell)
3. Run `py -m pip`
4. Run `py -m pip install --user pipenv`

At this point, you should be able to always run pipenv with `py -m pipenv`, but that's a little inconvenient. Read on for making it easier.

5. You'll see a message like this in the pipenv install output, but with a slightly different path:

```
add C:\Users\username\AppData\Roaming\Python\Python38\Scripts to your path
```

6. Select that path (not including any quotes around it), and copy it
7. Go to the Start menu, type "environment" and run the program  
`Edit Environment Variables`
8. In the System Properties popup, hit the `Advanced` tab, then `Environment Variables`
9. On the list of environment variables, doubleclick `Path`
10. Click `New`
11. Paste that path from step 5 into the new entry slot. Make sure there aren't any quotes around it.
12. Click `OK`, `OK`, `OK`.
13. Relaunch any shells you have open.

14. Now you should be able to just run `pip` and `pipenv` in Powershell without putting `py -m` in front of it.

## Pipenv official instructions

Install pipenv per these instructions

## Chocolatey

Install Chocolatey

Install Python 3 with Chocolatey

Install pipenv per these instructions

## WSL

If you're running Windows 10+, you might want to install the Windows Subsystem for Linux. This gives you a mini, integrated Linux system inside Windows. You then install and use Python from there.

1. Update Windows to latest if you haven't already.
2. Install WSL from [here](#).
3. Go to the Microsoft store and install Ubuntu.
4. Run Ubuntu to get a bash shell.
5. Make a new username and password. This is completely separate from your Windows username and password, but I made mine the same so that I wouldn't forget.
6. Upgrade the Ubuntu system. Run:

```
1 sudo apt-get update
2 sudo apt-get upgrade
```

Give your newly-entered password if prompted.

7. Running `python3 --version` should give you 3.6 or higher.
8. Run `pip install pipenv`.

If you've installed VS Code, add the "Remote WSL" extension. This way you can run `code` from within Ubuntu.

In the Ubuntu shell, you can run `explorer.exe .` in any directory to open a Windows Explorer window in that directory.

Also in Windows Explorer, you can put `\wsl$` in the URL bar to see your Ubuntu drive. (If it doesn't show up, relaunch your Ubuntu shell.)

If you run into trouble with the above, try the following:

1. Open cmd.exe as an administrator and start bash with `bash`

1. Type `Python -V` and `'Python3 -V'`

1. If one of these responds with `Python 3.6.8` use that command from now on

2. If neither response is `Python 3.6.8` but one is a higher version of Python, this means one of two things

1. If you have manually installed a higher version of Python, we recommend uninstalling it

2. If you have not, it is possible that Microsoft has updated WSL and you will need to adjust these instructions to accommodate

3. Otherwise, update Ubuntu:

1. `sudo apt-get update`

2. `sudo apt-get upgrade`

2. Repeat 2.1 above to determine if you should use `Python` or `Python3` when using Python. *Note:* inside the shell, you will always use `Python` as the command.

2. Make sure pip is installed for `Python 3`

1. `pip --version` and `pip3 --version`. One of these needs to respond with a version that has a version of Python 3 at the end of the line.

2. If you only have it for 2.7, you will need to install for 3 with:

1. `sudo apt update && sudo apt upgrade`

2. `sudo apt install python3-pip`

3. Check versions and commands again. You will likely need to use `pip3` for the next step, but it's possible it may be just `pip`. Use the one with the version associated with Python 3.6.8

3. Make sure pipenv is installed for Python 3 `python3 -m pipenv --version`

1. If not, install pipenv:

1. `sudo apt update && sudo apt upgrade` (if you didn't just do this above)

2. `pip3 install --user pipenv`
2. Check the version again
4. Try `pipenv shell`. If this fails, make sure that every reference in the error refers to Python
- 3.6. If not, review the above steps
  1. If the error does refer to 3.6:
    1. Confirm that `python --version` refers to 2.7.something
    2. Confirm that `/usr/bin/python3 --version` refers to 3.6.8
    3. `pipenv --three --python=` which `python3`` *NOTE* that there are backticks (`) around *which python3*
  4. This should create the shell forcing it to use 3.6.8

## Linux

Consult the documentation for your distribution.

# Virtual Environment Setup:

## Installing packages using pip and virtual environments

This guide discusses how to install packages using pip and a virtual environment manager: either `venv` for Python 3 or `virtualenv` for Python 2. These are the lowest-level tools for managing Python packages and are recommended if higher-level tools do not suit your needs.

### Note

This doc uses the term **package** to refer to a Distribution Package which is different from an Import Package that which is used to import modules in your Python source code.

### Installing pip

`pip` is the reference Python package manager. It's used to install and update packages. You'll need to make sure you have the latest version of `pip` installed. Unix/macOS

Debian and most other distributions include a `python-pip` package; if you want to use the Linux distribution-provided versions of `pip`, see [Installing pip/setuptools/wheel with Linux Package Managers](#).

You can also install `pip` yourself to ensure you have the latest version. It's recommended to use the system `pip` to bootstrap a user installation of `pip`:

```
1 python3 -m pip install --user --upgrade pip
2
3 python3 -m pip --version
```



[Copy to clipboard](#)

Afterwards, you should have the latest version of `pip` installed in your user site:

```
pip 21.1.3 from $HOME/.local/lib/python3.9/site-packages (python 3.9)
```



Windows

## Installing virtualenv

### Note

If you are using Python 3.3 or newer, the `venv` module is the preferred way to create and manage virtual environments. `venv` is included in the Python standard library and requires no additional installation. If you are using `venv`, you may skip this section.

`virtualenv` is used to manage Python packages for different projects. Using `virtualenv` allows you to avoid installing Python packages globally which could break system tools or other projects. You can install `virtualenv` using `pip`. Unix/macOS

```
python3 -m pip install --user virtualenv
```



Windows

## Creating a virtual environment

`venv` (for Python 3) and `virtualenv` (for Python 2) allow you to manage separate package installations for different projects. They essentially allow you to create a “virtual” isolated Python installation and install packages into that virtual installation. When you switch projects, you can simply create a new virtual environment and not have to worry about breaking the packages installed in the other environments. It is always recommended to use a virtual environment while developing Python applications.

To create a virtual environment, go to your project’s directory and run `venv`. If you are using Python 2, replace `venv` with `virtualenv` in the below commands. Unix/macOS

```
python3 -m venv env
```



Windows

The second argument is the location to create the virtual environment. Generally, you can just create this in your project and call it `env`.

`env` will create a virtual Python installation in the `env` folder.

#### Note

You should exclude your virtual environment directory from your version control system using `.gitignore` or similar.

## Activating a virtual environment

Before you can start installing or using packages in your virtual environment you'll need to *activate* it. Activating a virtual environment will put the virtual environment-specific `python` and `pip` executables into your shell's `PATH`. Unix/macOS

```
source env/bin/activate
```



Windows

You can confirm you're in the virtual environment by checking the location of your Python interpreter: Unix/macOS

```
which python
```



Windows

It should be in the `env` directory: Unix/macOS

```
.../env/bin/python
```



Windows

As long as your virtual environment is activated pip will install packages into that specific environment and you'll be able to import and use packages in your Python application.

## Leaving the virtual environment

If you want to switch projects or otherwise leave your virtual environment, simply run:

```
deactivate
```



[Copy to clipboard](#)

If you want to re-enter the virtual environment just follow the same instructions above about activating a virtual environment. There's no need to re-create the virtual environment.

## Installing packages

Now that you're in your virtual environment you can install packages. Let's install the Requests library from the Python Package Index (PyPI): Unix/macOS

```
python3 -m pip install requests
```



Windows

pip should download requests and all of its dependencies and install them:

```
1 Collecting requests
2 Using cached requests-2.18.4-py2.py3-none-any.whl
3 Collecting chardet<3.1.0,>=3.0.2 (from requests)
4 Using cached chardet-3.0.4-py2.py3-none-any.whl
5 Collecting urllib3<1.23,>=1.21.1 (from requests)
6 Using cached urllib3-1.22-py2.py3-none-any.whl
7 Collecting certifi>=2017.4.17 (from requests)
8 Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
9 Collecting idna<2.7,>=2.5 (from requests)
10 Using cached idna-2.6-py2.py3-none-any.whl
11 Installing collected packages: chardet, urllib3, certifi, idna, requests
12 Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4
```



Copy to clipboard

## Installing specific versions

pip allows you to specify which version of a package to install using version specifiers. For example, to install a specific version of `requests`:

```
python3 -m pip install requests==2.18.4
```



Windows

To install the latest `2.x` release of `requests`:

```
python3 -m pip install requests>=2.0.0,<3.0.0
```



Windows

To install pre-release versions of packages, use the `--pre` flag:Unix/macOS

```
python3 -m pip install --pre requests
```



Windows

## Installing extras

Some packages have optional extras. You can tell pip to install these by specifying the extra in brackets:Unix/macOS

```
python3 -m pip install requests[security]
```



Windows

## Installing from source

pip can install a package directly from source, for example:Unix/macOS

```
1 cd google-auth
2 python3 -m pip install .
```



Windows

Additionally, pip can install packages from source in development mode, meaning that changes to the source directory will immediately affect the installed package without needing to re-

install:Unix/macOS

```
python3 -m pip install --editable .
```



Windows

## Installing from version control systems

pip can install packages directly from their version control system. For example, you can install directly from a git repository:

```
git+https://github.com/GoogleCloudPlatform/google-auth-library-python.git#egg=google-auth
```



Copy to clipboard

For more information on supported version control systems and syntax, see pip's documentation on [VCS Support](#).

## Installing from local archives

If you have a local copy of a Distribution Package's archive (a zip, wheel, or tar file) you can install it directly with pip:Unix/macOS

```
python3 -m pip install requests-2.18.4.tar.gz
```



Windows

If you have a directory containing archives of multiple packages, you can tell pip to look for packages there and not to use the Python Package Index (PyPI) at all:Unix/macOS

```
python3 -m pip install --no-index --find-links=/local/dir/ requests
```



Windows

This is useful if you are installing packages on a system with limited connectivity or if you want to strictly control the origin of distribution packages.

## Using other package indexes

If you want to download packages from a different index than the Python Package Index (PyPI), you can use the `--index-url` flag:Unix/macOS

```
python3 -m pip install --index-url http://index.example.com/simple/ SomeProject
```



Windows

If you want to allow packages from both the Python Package Index (PyPI) and a separate index, you can use the `--extra-index-url` flag instead:Unix/macOS

```
python3 -m pip install --extra-index-url http://index.example.com/simple/ SomeP
```



Windows

## Upgrading packages

pip can upgrade packages in-place using the `--upgrade` flag. For example, to install the latest version of `requests` and all of its dependencies:Unix/macOS

```
python3 -m pip install --upgrade requests
```



Windows

## Using requirements files

Instead of installing packages individually, pip allows you to declare all dependencies in a Requirements File. For example you could create a `requirements.txt` file containing:

```
1 requests==2.18.4
2 google-auth==1.1.0
```



Copy to clipboard

And tell pip to install all of the packages in this file using the `-r` flag:Unix/macOS

```
python3 -m pip install -r requirements.txt
```



Windows

## Freezing dependencies

Pip can export a list of all installed packages and their versions using the `freeze` command:Unix/macOS

```
python3 -m pip freeze
```



Windows

Which will output a list of package specifiers such as:

```
1 cachetools==2.0.1
2 certifi==2017.7.27.1
3 chardet==3.0.4
4 google-auth==1.1.1
5 idna==2.6
6 pyasn1==0.3.6
7 pyasn1-modules==0.1.4
8 requests==2.18.4
9 rsa==3.4.2
10 six==1.11.0
11 urllib3==1.22
```



[Copy to clipboard](#)

This is useful for creating Requirements Files that can re-create the exact versions of all packages installed in an environment.

# Installing stand alone command line tools

Many packages provide command line applications. Examples of such packages are mypy, flake8, black, and Pipenv.

Usually you want to be able to access these applications from anywhere on your system, but installing packages and their dependencies to the same global environment can cause version conflicts and break dependencies the operating system has on Python packages.

pipx solves this by creating a virtual environment for each package, while also ensuring that its applications are accessible through a directory that is on your `$PATH`. This allows each package to be upgraded or uninstalled without causing conflicts with other packages, and allows you to safely run the applications from anywhere.

## Note

pipx only works with Python 3.6+.

pipx is installed with pip:Unix/macOS

```
1 python3 -m pip install --user pipx
2 python3 -m pipx ensurepath
```



Windows

## Note

`ensurepath` ensures that the application directory is on your `$PATH`. You may need to restart your terminal for this update to take effect.

Now you can install packages with `pipx install` and run the package's applications(s) from anywhere.

```
1 $ pipx install PACKAGE
2 $ PACKAGE_APPLICATION [ARGS]
```



Copy to clipboard

For example:

```
1 $ pipx install cowsay
2 installed package cowsay 2.0, Python 3.6.2+
3 These binaries are now globally available
4 - cowsay
5 done! ☆ ┌ ☆
6 $ cowsay moo
7
8 < moo >
9 ===
10 \
11 \
12 ^__^
13 (oo)_____
14 (__)\)\/\
15 || ||
```



Copy to clipboard

To see a list of packages installed with pipx and which applications are available, use

`pipx list`:

```
1 $ pipx list
2 venvs are in /Users/user/.local/pipx/venvs
3 symlinks to binaries are in /Users/user/.local/bin
4 package black 18.9b0, Python 3.6.2+
5 - black
6 - blackd
7 package cowsay 2.0, Python 3.6.2+
8 - cowsay
9 package mypy 0.660, Python 3.6.2+
10 - dmypy
11 - mypy
```

```
12 - stubgen
13 package nox 2018.10.17, Python 3.6.2+
14 - nox
15 - tox-to-nox
```



[Copy to clipboard](#)

To upgrade or uninstall a package:

```
1 pipx upgrade PACKAGE
2 pipx uninstall PACKAGE
```



[Copy to clipboard](#)

pipx can be upgraded or uninstalled with pip:Unix/macOS

```
1 python3 -m pip install -U pipx
2 python3 -m pip uninstall pipx
```



Windows

pipx also allows you to install and run the latest version of an application in a temporary, ephemeral environment. For example:

```
pipx run cowsay moo
```



[Copy to clipboard](#)

To see the full list of commands pipx offers, run:

```
pipx --help
```



[Copy to clipboard](#)

You can learn more about pipx at <https://pypa.github.io/pipx/>.

# Pip

To get started with using pip, you should install Python on your system.

---

## Ensure you have a working pip¶

As a first step, you should check that you have a working Python with pip installed. This can be done by running the following commands and making sure that the output looks similar.

Linux

```
$ python -version Python 3.N.N $ python -m pip -version pip X.Y.Z from ... (python 3.N.N)
```

MacOS

```
$ python -version Python 3.N.N $ python -m pip -version pip X.Y.Z from ... (python 3.N.N)
```

Windows

```
C:> py -version Python 3.N.N C:> py -m pip -version pip X.Y.Z from ... (python 3.N.N)
```

If that worked, congratulations! You have a working pip in your environment.

If you got output that does not look like the sample above, please read the [Installation](#) page. It provides guidance on how to install pip within a Python environment that doesn't have it.

---

## Common tasks¶

### Install a package¶

Linux

```
$ python -m pip install sampleproject [...] Successfully installed sampleproject
```

MacOS

```
$ python -m pip install sampleproject [...] Successfully installed sampleproject
```

Windows

```
C:> py -m pip install sampleproject [...] Successfully installed sampleproject
```

By default, pip will fetch packages from Python Package Index, a repository of software for the Python programming language where anyone can upload packages.

## Install a package from GitHub¶

Linux

```
$ python -m pip install git+https://github.com/pypa/sampleproject.git@main [...] Successfully installed sampleproject
```

MacOS

```
$ python -m pip install git+https://github.com/pypa/sampleproject.git@main [...] Successfully installed sampleproject
```

Windows

```
C:> py -m pip install git+https://github.com/pypa/sampleproject.git@main [...] Successfully installed sampleproject
```

See VCS Support for more information about this syntax.

## Install a package from a distribution file¶

pip can install directly from distribution files as well. They come in 2 forms:

- source distribution (usually shortened to “sdist”)
- wheel distribution (usually shortened to “wheel”)

Linux

```
$ python -m pip install sampleproject-1.0.tar.gz [...] Successfully installed sampleproject $
python -m pip install sampleproject-1.0-py3-none-any.whl [...] Successfully installed
sampleproject
```

MacOS

```
$ python -m pip install sampleproject-1.0.tar.gz [...] Successfully installed sampleproject $
python -m pip install sampleproject-1.0-py3-none-any.whl [...] Successfully installed
sampleproject
```

Windows

```
C:> py -m pip install sampleproject-1.0.tar.gz [...] Successfully installed sampleproject C:> py -m
pip install sampleproject-1.0-py3-none-any.whl [...] Successfully installed sampleproject
```

## Install multiple packages using a requirements file¶

Many Python projects use `requirements.txt` files, to specify the list of packages that need to be installed for the project to run. To install the packages listed in that file, you can run:

Linux

```
$ python -m pip install -r requirements.txt [...] Successfully installed sampleproject
```

MacOS

```
$ python -m pip install -r requirements.txt [...] Successfully installed sampleproject
```

Windows

```
C:> py -m pip install -r requirements.txt [...] Successfully installed sampleproject
```

## Upgrade a package¶

Linux

```
$ python -m pip install --upgrade sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

MacOS

```
$ python -m pip install --upgrade sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

Windows

```
C:> py -m pip install --upgrade sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

## Uninstall a package¶

Linux

```
$ python -m pip uninstall sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

MacOS

```
$ python -m pip uninstall sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

Windows

```
C:> py -m pip uninstall sampleproject Uninstalling sampleproject: [...] Proceed (y/n)? y
Successfully uninstalled sampleproject
```

---

## Next Steps¶

It is recommended to learn about what virtual environments are and how to use them. This is covered in the “Installing Packages” tutorial on [packaging.python.org](http://packaging.python.org).

WK-19

# D3

## Objective 01 - Describe the properties of a binary tree and the properties of a "perfect" tree

### Overview

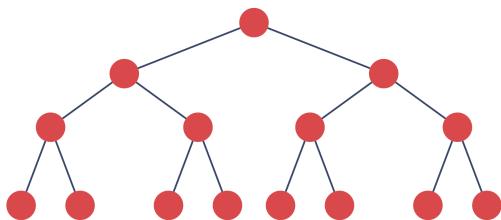
There are lots of different types of tree data structures. A binary tree is a specific type of tree. It is called a binary tree because each node in the tree can only have a maximum of two child nodes. It is common for a node's children to be called either `left` or `right`.

Here is an example of what a class for a binary tree node might look like:

```
1 class BinaryTreeNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
```

### Follow Along

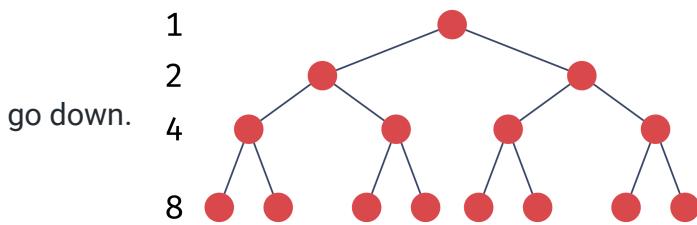
With this simple class, we can now build up a structure that could be visualized like so:



### "Perfect" Trees

A "perfect" tree has all of its levels full. This means that there are not any missing nodes in each level.

"Perfect" trees have specific properties. First, the quantity of each level's nodes doubles as you



Second, the quantity of the last level's nodes is the same as the quantity of all the other nodes plus one.

These properties are useful for understanding how to calculate the *height* of a tree. The height of a tree is the number of levels that it contains. Based on the properties outlined above, we can deduce that we can calculate the tree's height with the following formula:  $\log_2(n + 1) = h$

In the formula above,  $n$  is the total number of nodes. If you know the tree's height and want to calculate the total number of nodes, you can do so with the following formula:  $n = 2^h - 1$

We can represent the relationship between a perfect binary tree's total number of nodes and its height because of the properties outlined above.

## Challenge

1. Calculate how many levels a perfect binary tree has given that the total number of nodes is 127.
2. Calculate the total number of nodes on a perfect binary tree, given that the tree's height is 8.

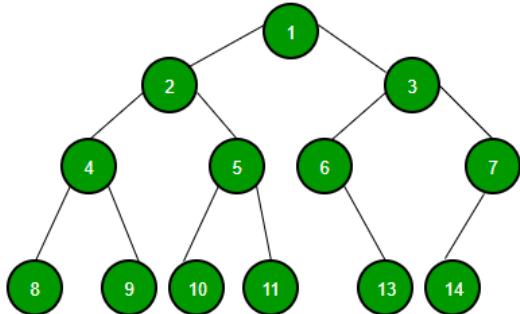
## Additional Resources

- [https://en.wikipedia.org/wiki/Binary\\_tree](https://en.wikipedia.org/wiki/Binary_tree) (Links to an external site.)
- <https://www.geeksforgeeks.org/binary-tree-data-structure/>
- **Binary Tree Data Structure**

---

## Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Output: `foreach test case, print the answer` \*\*Constraints:  $1 \leq T \leq 52$ .  $1 \leq N \leq 105$ .  $1 \leq arr[i] \leq 105$

Example: `Input: 24 12 34 12 34 41 15 12 12 11 22 33 44 5` \*\*Output: `21`

% Given a tree with N nodes rooted at 1. Each node labeled with a value  $arr[i]$ . The task is to find the absolute difference between the sum of values of nodes at even level and odd level % Note: All the nodes are numbered from 1 to N. %\*\*Input:\*\*1. The first line of the input contains a single integer \*\*T\*\* denoting the number of test cases. The description of T test cases follows.2. The first line of each test case contains a single integer \*\*N.\*\*3. The next line contains N space-separated positive integers represents the node value.4. Next  $N-1$  lines contain two space-separated integers  $u$  and  $v$ , represents an edge in between them Output: For each test case, print the answer \*\*Constraints:  $1 \leq T \leq 52$ .  $1 \leq N \leq 105$ .  $1 \leq arr[i] \leq 105$

Example: `Input: 24 1 2 3 4 1 2 3 4 4 15 1 2 1 2 11 22 33 44 5` \*\*Output: `21`

Output: `foreach test case, print the answer` \*\*Constraints:  $1 \leq T \leq 52$ .  $1 \leq N \leq 105$ .  $1 \leq arr[i] \leq 105$

Example: `Input: 24 12 34 12 34 41 15 12 12 11 22 33 44 5` \*\*Output: `21`

## Tree and Level

Given a tree with N nodes rooted at 1. Each node labeled with a value  $arr[i]$ . The task is to find the absolute difference between the sum of values of nodes at even level and odd level Note: All the nodes are numbered from 1 to N. Input:1. The first line of the input contains a single integer \*\*T\*\* denoting the number of test cases. The description of T test cases follows.2. The first line of each test case contains a single integer N.3. The next line contains N space-

separated positive integers represents the node value.4. Next N-1 lines contain two space-separated integers u and v, represents an edge in between them Output: For each test case, print the answer Constraints:1. 1  $\leq$  T  $\leq$  52. 1  $\leq$  N  $\leq$  1053. 1  $\leq$  arr[i]  $\leq$  105Example:Input:241 2  
3 41 23 44 151 2 1 2 11 22 33 44 5 Output:21

**Output:**21

```
1 import math
2 t = int(input())
3 for _ in range(t):
4 n = int(input())
5 graph = [[] for _ in range(n + 1)]
6 value = list(map(int, input().strip().split(" ")))
7 for _ in range(n - 1):
8 u, v = tuple(map(int, input().strip().split(" ")))
9 graph[u].append(v)
10 graph[v].append(u)
11 q = queue.Queue()
12 q.put(1)
13 level = [None] * (n + 1)
14 visited = [False] * (n + 1)
15 level[1] = 0
16 visited[1] = True
17 even_sum = 0
18 odd_sum = 0
19 while(not q.empty()):
20 node = q.get()
21 if level[node] % 2 == 0:
22 even_sum += value[node - 1]
23 else:
24 odd_sum += value[node - 1]
25
26 for edge_vertex in graph[node]:
27 if visited[edge_vertex]:
28 continue
29 q.put(edge_vertex)
30 visited[edge_vertex] = True
31 new_level = level[node] + 1
32 level[edge_vertex] = new_level
33
34 print(abs(even_sum - odd_sum))
```

OR

```
1 def bfs(tree,arr,n):
```

```

2 vis=[False for i in range(n)]
3 Q=deque()
4 Q.append((0,0))
5 vis[0]=True
6 even=0
7 odd=0
8
9 while Q:
10 u,l=Q.popleft()
11 if l%2==0:
12 even+=arr[u]
13 elif l%2!=0:
14 odd+=arr[u]
15
16 for v in tree[u]:
17 if vis[v]==False:
18 Q.append((v,l+1))
19 vis[v]=True
20
21 return abs(even-odd)
22
23 def main():
24 test=int(input())
25 for t in range(test):
26 n=int(input())
27 arr=list(map(int,input().split()))
28 tree=defaultdict(list)
29 for i in range(n-1):
30 u,v=map(int,input().split())
31 u-=1
32 v-=1
33 tree[u].append(v)
34 tree[v].append(u)
35
36 ans=bfs(tree,arr,n)
37 print(ans)
38
39 main()

```

```

1 def bfs(d):
2 visited = {1: 1}
3 que = [1]
4 while len(que):
5 elem = que[0]
6 level = visited[elem]
7
8 if elem in d:
9 for i in d[elem]:
10 if i not in visited:
11 visited[i] = level + 1

```

```

12 que.append(i)
13
14 que.pop(0)
15
16 return visited
17
18 for _ in range(int(input())):
19 n = int(input())
20 l = list(map(int, input().split()))
21
22 d = {}
23 for i in range(n-1):
24 a, b = map(int, input().split())
25
26 if a not in d:
27 d[a] = [b]
28 else:
29 d[a].append(b)
30
31 if b not in d:
32 d[b] = [a]
33 else:
34 d[b].append(a)
35
36 levels = bfs(d)
37
38 x, y = 0, 0
39 for i in levels:
40 if levels[i] % 2:
41 x += l[i-1]
42 else:
43 y += l[i-1]
44
45 print(abs(x - y))

```

Office Hours:

---

## Objective 01 - Describe the properties of a binary tree and the properties of a "perfect" tree

### Overview

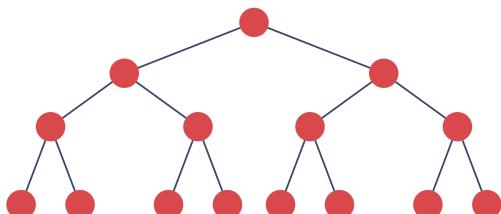
There are lots of different types of tree data structures. A binary tree is a specific type of tree. It is called a binary tree because each node in the tree can only have a maximum of two child nodes. It is common for a node's children to be called either `left` or `right`. Here is an example of what a class for a binary tree node might look like:

exit: Ctrl+←

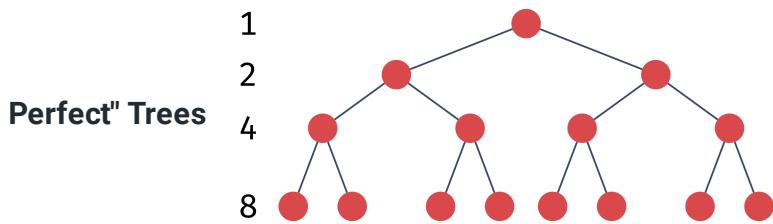
```
1 class BinaryTreeNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
```

## Follow Along

With this simple class, we can now build up a structure that could be visualized like so:



[https://tk-assets.lambdaschool.com/c00c8f45-abff-4c3a-b29b-92631b5ac88e\\_binary-tree-example.001.png](https://tk-assets.lambdaschool.com/c00c8f45-abff-4c3a-b29b-92631b5ac88e_binary-tree-example.001.png)  
Enter a caption for this image (optional)



[https://tk-assets.lambdaschool.com/36747e43-d96d-40c9-b8ab-d318f6da8aed\\_binary-tree-example-levels.001.png](https://tk-assets.lambdaschool.com/36747e43-d96d-40c9-b8ab-d318f6da8aed_binary-tree-example-levels.001.png) Enter a caption for this image (optional)

A "perfect" tree has all of its levels full. This means that there are not any missing nodes in each level. "Perfect" trees have specific properties. First, the quantity of each level's nodes

doubles as you go down. Second, the quantity of the last level's nodes is the same as the quantity of all the other nodes plus one. These properties are useful for understanding how to calculate the

*height*

of a tree. The height of a tree is the number of levels that it contains. Based on the properties outlined above, we can deduce that we can calculate the tree's height with the following formula:  $\lceil \log_2(n+1) \rceil = h$

[https://i.upmath.me/svg/log\\_2\(n%2B1\) %3D h](https://i.upmath.me/svg/log_2(n%2B1) %3D h)

) In the formula above,

n

is the total number of nodes. If you know the tree's height and want to calculate the total number of nodes, you can do so with the following formula:  $n = 2^h - 1$

We can represent the relationship between a perfect binary tree's total number of nodes and its height because of the properties outlined above.

## **Challenge**

1. Calculate how many levels a perfect binary tree has given that the total number of nodes is 127.
  2. Calculate the total number of nodes on a perfect binary tree, given that the tree's height is 8.
- Additional Resources

- [https://en.wikipedia.org/wiki/Binary\\_tree](https://en.wikipedia.org/wiki/Binary_tree) (Links to an external site.)
  - <https://www.geeksforgeeks.org/binary-tree-data-structure/> (Links to an external site.)
- 

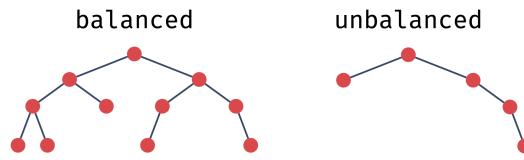
## **Objective 02 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a binary search tree**

### **Overview**

Just like a binary tree is a specific type of tree, a binary search tree (BST) is a specific type of binary tree. A binary search tree is just like a binary tree, except it follows specific rules about how it orders the nodes contained within it. For each node in the BST, all the nodes to the left are smaller, and all the nodes to the right of it are larger. We can call a binary search tree

balanced if the heights of its left and right subtrees differ by at most one, and both of the

subtrees are also balanced.



[https://tk-assets.lambdaschool.com/f84f26b9-09f3-48e0-a4c6-a51740d9c083\\_binary-tree-example-balanced-unbalanced.001.png](https://tk-assets.lambdaschool.com/f84f26b9-09f3-48e0-a4c6-a51740d9c083_binary-tree-example-balanced-unbalanced.001.png) Enter a caption for this image (optional)

## Follow Along

### Time and Space Complexity

### Lookup

If a binary search tree is balanced, then a lookup operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is unbalanced, the time complexity can be linear ( $O(n)$ ) in the worst possible case (virtually a linear chain of nodes will have all the nodes on one side of the tree).

### Insert

If a binary search tree is balanced, then an insertion operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is entirely unbalanced, then the time complexity is linear ( $O(n)$ ) in the worst case.

## Delete

If a binary search tree is balanced, then a deletion operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is entirely unbalanced, then the time complexity is linear ( $O(n)$ ) in the worst case.

## Space

The space complexity of a binary search tree is linear ( $O(n)$ ). Each node in the binary search tree will take up space in memory.

## Strengths

One of the main strengths of a BST is that it is sorted by default. You can pull out the data in order by using an in-order traversal. BSTs also have efficient searches ( $O(\log n)$ ). They have the same efficiency for their searches as a sorted array; however, BSTs are faster with insertions and deletions. In the average-case, dictionaries have more efficient operations than BSTs, but a BST has more efficient operations in the worst-case.

## Weaknesses

The primary weakness of a BST is that they only have efficient operations if they are balanced. The more unbalanced they are, the worse the efficiency of their operations gets. Another

weakness is that they are don't have stellar efficiency in any one operation. They have good efficiency for a lot of different operations. So, they are more of a general-purpose data structure. If you want to learn more about trees that automatically rearrange their nodes to remain balanced, look into AVL trees (Links to an external site.) or Red-Black trees (Links to an external site.)

## Challenge

1. In your own words, explain why an unbalanced binary search tree's performance becomes degraded. Additional Resources

- <https://www.geeksforgeeks.org/binary-search-tree-data-structure/> (Links to an external site.)
  - [https://en.wikipedia.org/wiki/Binary\\_search\\_tree](https://en.wikipedia.org/wiki/Binary_search_tree) (Links to an external site.)
- 

## Objective 03 - Construct a binary search tree that can perform basic operations with a logarithmic time complexity

### Overview

To create a binary search tree, we need to define two different classes: one for the nodes that will make up the binary search tree and another for the tree itself.

## Follow Along

Let's start by creating a `BSTNode` class. An instance of `BSTNode` should have a `value`, a `right` node, and a `left` node.

exit: `Ctrl+←`

```
1 class BSTNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
```

Now that we have our basic `BSTNode` class defined with an initialization method let's define our `BST` class. This class will have an initialization method and an `insert` method.

exit: `Ctrl+←`

```
1 class BST:
2 def __init__(self, value):
3 self.root = BSTNode(value)
4 def insert(self, value):
5 self.root.insert(value)
```

Notice that our `BST` class expects each `BSTNode` to have an `insert` method available on an instance object. But, we haven't yet added an `insert` method on the `BSTNode` class. Let's do that now.

exit: `Ctrl+←`

```
1 class BSTNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
6 def insert(self, value):
7 if value < self.value:
8 if self.left is None:
9 self.left = BSTNode(value)
10 else:
11 self.left.insert(value)
12 else:
13 if self.right is None:
14 self.right = BSTNode(value)
15 else:
16 self.right.insert(value)
```

Now that we can insert nodes into our binary search tree let's define a `search` method that can lookup values in our binary search tree.

exit: Ctrl+←

```
1 class BST:
2 def __init__(self, value):
3 self.root = BSTNode(value)
4 def insert(self, value):
5 self.root.insert(value)
6 def search(self, value):
7 self.root.search(value)
```

Our `BST` class expects there to be a `search` method available on the `BSTNode` instance stored at the root. Let's go ahead and define that now.

exit: Ctrl+←

```
1 class BSTNode:
2 def __init__(self, value):
```

```

3 self.value = value
4 self.left = None
5 self.right = None
6 def insert(self, value):
7 if value < self.value:
8 if self.left is None:
9 self.left = BSTNode(value)
10 else:
11 self.left.insert(value)
12 else:
13 if self.right is None:
14 self.right = BSTNode(value)
15 else:
16 self.right.insert(value)
17 def search(self, target):
18 if self.value == target:
19 return self
20 elif target < self.value:
21 if self.left is None:
22 return False
23 else:
24 return self.left.search(target)
25 else:
26 if self.right is None:
27 return False
28 else:
29 return self.right.search(target)

```

## Challenge

To implement a `delete` operation on our `BST` and `BSTNode` classes, we must consider three cases: 1. If the `BSTNode` to be deleted is a leaf (has no children), we can remove that node from the tree. 2. If the `BSTNode` to be deleted has only one child, we copy the child node to be deleted and delete it. 3. If the `BSTNode` to be deleted has two children, we have to find the "in-order successor". The "in-order successor" is the next highest value, the node that has the minimum value in the right subtree. Given the above information, can you write pseudocode for a method that can find the *minimum value* of all the nodes within a tree or subtree?

## **Additional Resources**

- <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/> (Links to an external site.)
- <https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/> (Links to an external site.)

# D1 - Linked Lists & BigO

 [Linked List Pdf Download](#)

10-linkedlist.pdf - 960KB



GitHub -  
bgoonz/DATA\_STRUC\_PYTHON\_NOTES

[https://github.com/bgoonz/DATA\\_STRUC\\_PYTHON\\_NOTES](https://github.com/bgoonz/DATA_STRUC_PYTHON_NOTES)

bgoonz/  
**DATA\_STRUC\_PYTHON\_...**



 4  
Contributors

 1  
Issue

 4  
Stars

 3  
Forks



---

**Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a linked list**

## Overview

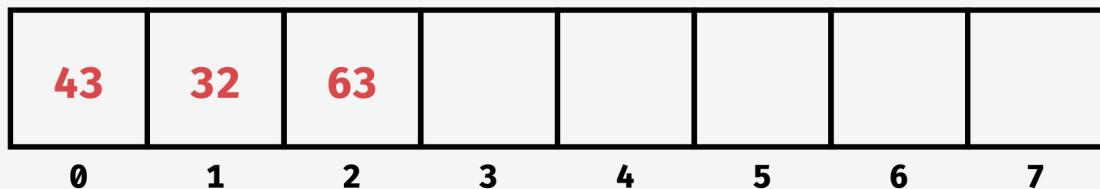
What is a linked list, and how is it different from an array? How efficient or inefficient are its operations? What are its strengths and weaknesses? How can I construct and interact with a linked list? By the end of this objective, you will be able to answer all of these questions confidently.

## Follow Along

### Basic Properties of a Linked List

A linked list is a simple, linear data structure used to store a collection of elements. Unlike an array, each element in a linked list does not have to be stored contiguously in memory.

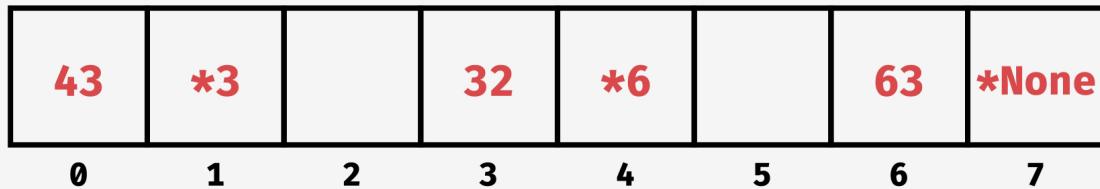
For example, in an array, each element of the list `[43, 32, 63]` is stored in memory like so:



[https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767\\_arrays-stored-in-memory.001.jpeg](https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767_arrays-stored-in-memory.001.jpeg)

`43` is the first item in the collection and is therefore stored in the first slot. `32` is the second item and is stored immediately next to `43` in memory. This pattern continues on and on.

In a linked list, each element of the list could be stored like so:



[https://tk-assets.lambdaschool.com/72151497-7a5e-4940-835c-d8beb9c88922\\_linked-list-in-memory.001.jpeg](https://tk-assets.lambdaschool.com/72151497-7a5e-4940-835c-d8beb9c88922_linked-list-in-memory.001.jpeg)

You can see here that the elements can be spaced out in memory. Because the elements are not stored contiguously, each element in memory must contain information about the next element in the list. The first item stores the data `43` and the location in memory (`*3`) for the next item in the list. This example is simplified; the second item in the list `32` could be located anywhere in memory. It could even come before the first item in memory.

You might also be wondering what types of data can be stored in a linked list. Pretty much any type of data can be stored in a linked list. Strings, numbers, booleans, and other data structures can be stored. You should not feel limited using a linked list based on what type of data you are trying to store.

Are the elements in a linked list sorted or unsorted? The elements in a linked list can be either sorted or unsorted. There is nothing about the data structure that forces the elements to be sorted or unsorted. You cannot determine if a linked list's elements are sorted by determining they are stored in a linked list.

What about duplicates? Can a linked list contain them? Linked lists can contain duplicates. There is nothing about the linked list data structure that would prevent duplicates from being stored. When you encounter a linked list, you should know that it can contain duplicates.

Are there different types of linked lists? If so, what are they? There are three types of linked lists: singly linked list (SLL), doubly linked list (DLL), and circular linked list. All linked lists are made up of nodes where each node stores the data and also information about other nodes in the linked list.

Each singly linked list node stores the data and a pointer where the next node in the list is located. Because of this, you can only navigate in the forward direction in a singly linked list. To traverse an SLL, you need a reference to the first node called the head. From the head of the list, you can visit all the other nodes using the next pointers.

The difference between an SLL and a doubly linked list (DLL) is that each node in a DLL also stores a reference to the previous item. Because of this, you can navigate forward and backward in the list. A DLL also usually stores a pointer to the last item in the list (called the tail).

A Circular Linked List links the last node back to the first node in the list. This linkage causes a circular traversal; when you get to the end of the list, the next item will be back at the beginning of the list. Each type of linked list is similar but has small distinctions. When working with linked lists, it's essential to know what type of linked list.

## Time and Space Complexity

### Lookup

To look up an item by index in a linked list is linear time ( $O(n)$ ). To traverse through a linked list, you have to start with the head reference to the node and then follow each subsequent pointer to the next item in the chain. Because each item in the linked list is not stored contiguously in memory, you cannot access a specific index of the list using simple math. The distance in memory between one item and the next is varied and unknown.

### Append

Adding an item to a linked list is constant time ( $O(1)$ ). We always have a reference point to the tail of the linked list, so we can easily insert an item after the tail.

### Insert

In the worst case, inserting an item in a linked list is linear time ( $O(n)$ ). To insert an item at a specific index, we have to traverse — starting at the head — until we reach the desired index.

## Delete

In the worst case, deleting an item in a linked list is linear time ( $O(n)$ ). Just like insertion, deleting an item at a specific index means traversing the list starting at the head.

## Space

The space complexity of a linked list is linear ( $O(n)$ ). Each item in the linked list will take up space in memory.

## Strengths of a Linked List

The primary strength of a linked list is that operations on the linked list's ends are fast. This is because the linked list always has a reference to the head (the first node) and the tail (the last node) of the list. Because it has a reference, doing anything on the ends is a constant time operation ( $O(1)$ ) no matter how many items are stored in the linked list. Additionally, just like a dynamic array, you don't have to set a capacity to a linked list when you instantiate it. If you don't know the size of the data you are storing, or if the amount of data is likely to fluctuate, linked lists can work well. One benefit over a dynamic array is that you don't have doubling appends. This is because each item doesn't have to be stored contiguously; whenever you add an item, you need to find an open spot in memory to hold the next node.

## Weaknesses of a Linked List

The main weakness of a linked list is not efficiently accessing an "index" in the middle of the list. The only way that the linked list can get to the seventh item in the linked list is by going to the head node and then traversing one node at a time until you arrive at the seventh node. You can't do simple math and jump from the first item to the seventh.

## What data structures are built on linked lists?

Remember that linked lists have efficient operations on the ends (head and tail). There are two structures that only operate on the ends; queues and stacks. So, most queue or stack implementations use a linked list as their underlying data structure.

## Why is a linked list different than an array? What problem does it solve?

We can see the difference between how a linked list and an array are stored in memory, but why is this important? Once you see the problem with the way arrays are stored in memory, the benefits of a linked list become clearer.

The primary problem with arrays is that they hold data contiguously in memory. Remember that having the data stored contiguously is the feature that gives them quick lookups. If I know where the first item is stored, I can use simple math to figure out where the fifth item is stored. The reason that this is a problem is that it means that when you create an array, you either have to know how much space in memory you need to set aside, or you have to set aside a bunch of extra memory that you might not need, just in case you do need it. In other words, you can be space-efficient by only setting aside the memory you need at the moment. But, in doing that, you are setting yourself up for low time efficiency if you run out of room and need to copy all of your elements to a newer, bigger array.

With a linked list, the elements are not stored side-by-side in memory. Each element can be stored anywhere in memory. In addition to storing the data for that element, each element also stores a pointer to the memory location of the next element in the list. The elements in a linked list do not have an index. To get to a specific element in a linked list, you have to start at the head of the linked list and work your way through the list, one element at a time, to reach the specific element you are searching for. Now you can see how a linked list solves some of the problems that the array data structure has.

### **How do you represent a linked list graphically and in Python code?**

Let's look at how we can represent a singly linked list graphically and in Python code. Seeing a singly linked list represented graphically and in code can help you understand it better.

How do you represent a singly linked list graphically? Let's say you wanted to store the numbers 1, 2, and 3. You would need to create three nodes. Then, each of these nodes would be linked together using the pointers.



[https://tk-assets.lambdaschool.com/baa6486b-9322-481e-95be-c660640c4966\\_linked-list-graphical-representation.001.jpeg](https://tk-assets.lambdaschool.com/baa6486b-9322-481e-95be-c660640c4966_linked-list-graphical-representation.001.jpeg)

Notice that the last element or node in the linked list does not have a pointer to any other node. This fact is how you know you are at the end of the linked list.

What does a singly linked list implementation look like in Python? Let's start by writing a `LinkedListNode` class for each element in the linked list.

```
1 class LinkedListNode:
2 def __init__(self, data=None, next=None):
3 self.data = data
4 self.next = next
```

Now, we need to build out the class for the `LinkedList` itself:

```
1 class LinkedList:
2 def __init__(self, head=None):
3 self.head = head
```

Our class is super simple so far and only includes an initialization method. Let's add an `append` method so that we can add nodes to the end of our list:

```
1 class LinkedList:
2 def __init__(self, head=None):
3 self.head = head
4
5 def append(self, data):
6 new_node = LinkedListNode(data)
7
8 if self.head:
9 current = self.head
10
11 while current.next:
12 current = current.next
13
14 current.next = new_node
15 else:
16 self.head = new_node
```

Now, let's use our simple class definitions for `LinkedListNode` and `LinkedList` to create a linked list of elements `1`, `2`, and `3`.

```
1 >>> a = LinkedListNode(1)
2 >>> my_ll = LinkedList(a)
3 >>> my_ll.append(2)
4 >>> my_ll.append(3)
5 >>> my_ll.head.data
6 1
7 >>> my_ll.head.next.data
8 2
9 >>> my_ll.head.next.next.data
10 3
11 >>>
```

You must be able to understand and interact with linked lists. You now know the basic properties and types of linked lists, what makes a linked list different from an array, what problem it solves, and how to represent them both graphically and in code. You now know enough about linked lists that you should be able to solve algorithmic code challenges that require a basic understanding of linked lists.

## Challenge

1. Draw out a model of a singly-linked list that stores the following integers in order:

3,2,6,5,7,9 .

2. Draw out a model of a doubly-linked list that stores the following integers in order:

5,2,6,4,7,8 .

---

## Additional Resources



[https://www.youtube.com/watch?v=njTh\\_OwMljA](https://www.youtube.com/watch?v=njTh_OwMljA)

[https://www.youtube.com/watch?v=njTh\\_OwMljA](https://www.youtube.com/watch?v=njTh_OwMljA)



Data Structures: Linked Lists

[https://youtu.be/njTh\\_OwMljA](https://youtu.be/njTh_OwMljA)



Linked Lists Pdf

10-linkedlist.pdf - 960KB

```
1 # -*- coding: utf-8 -*-
2 """Linked Lists.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at
7 https://colab.research.google.com/drive/17MD2e14fi7n95HTvy1K_ttM0FZSYnLmm
8
9 # Linked Lists
```

```

10 - Non Contiguous abstract Data Structure
11 - Value (can be any value for our use we will just use numbers)
12 - Next (A pointer or reference to the next node in the list)
13
14 ...
15 L1 = Node(34)
16 L1.next = Node(45)
17 L1.next.next = Node(90)
18
19 # while the current node is not none
20 # do something with the data
21 # traverse to next node
22
23 L1 = [34]-> [45]-> [90] -> None
24
25 Node(45)
26 Node(90)
27
28 ...
29 """
30
31 class LinkedListNode:
32 """
33 Simple Singly Linked List Node Class
34 value -> int
35 next -> LinkedListNode
36 """
37 def __init__(self, value):
38 self.value = value
39 self.next = None
40
41 def add_node(self, value):
42 # set current as a ref to self
43 current = self
44 # thile there is still more nodes
45 while current.next is not None:
46 # traverse to the next node
47 current = current.next
48 # create a new node and set the ref from current.next to the new node
49 current.next = LinkedListNode(value)
50
51 def insert_node(self, value, target):
52 # create a new node with the value provided
53 new_node = LinkedListNode(value)
54 # set a ref to the current node
55 current = self
56 # while the current nodes value is not the target
57 while current.value != target:
58 # traverse to the next node
59 current = current.next
60 # set the new nodes next pointer to point toward the current nodes next po
61 new_node.next = current.next
62 # set the current nodes next to point to the new node

```

```

63 current.next = new_node
64
65 ll_storage = []
66 L1 = LinkedListNode(34)
67 L1.next = LinkedListNode(45)
68 L1.next.next = LinkedListNode(90)
69
70 def print_ll(linked_list_node):
71 current = linked_list_node
72 while current is not None:
73 print(current.value)
74 current = current.next
75
76 def add_to_ll_storage(linked_list_node):
77 current = linked_list_node
78 while current is not None:
79 ll_storage.append(current)
80 current = current.next
81
82 L1.add_node(12)
83 print_ll(L1)
84 L1.add_node(24)
85 print()
86 print_ll(L1)
87 print()
88 L1.add_node(102)
89 print_ll(L1)
90 L1.insert_node(123, 90)
91 print()
92 print_ll(L1)
93 L1.insert_node(678, 34)
94 print()
95 print_ll(L1)
96 L1.insert_node(999, 102)
97 print()
98 print_ll(L1)
99
100 """# CODE 9571"""
101
102 class LinkedListNode:
103 """
104 Simple Doubly Linked List Node Class
105 value -> int
106 next -> LinkedListNode
107 prev -> LinkedListNode
108 """
109 def __init__(self, value):
110 self.value = value
111 self.next = None
112 self.prev = None
113
114 """
115 Given a reference to the head node of a singly-linked list, write a function

```

```

116 that reverses the linked list in place. The function should return the new head
117 of the reversed list.
118 In order to do this in O(1) space (in-place), you cannot make a new list, you
119 need to use the existing nodes.
120 In order to do this in O(n) time, you should only have to traverse the list
121 once.
122 *Note: If you get stuck, try drawing a picture of a small linked list and
123 running your function by hand. Does it actually work? Also, don't forget to
124 consider edge cases (like a list with only 1 or 0 elements).*
125 cn p
126 None [1] -> [2] ->[3] -> None
127
128
129 - setup a current variable pointing to the head of the list
130 - set up a prev variable pointing to None
131 - set up a next variable pointing to None
132
133 - while the current ref is not none
134 - set next to the current.next
135 - set the current.next to prev
136 - set prev to current
137 - set current to next
138
139 - return prev
140
141 """
142 class LinkedListNode():
143 def __init__(self, value):
144 self.value = value
145 self.next = None
146
147 def reverse(head_of_list):
148 current = head_of_list
149 prev = None
150 next = None
151
152 while current:
153 next = current.next
154 current.next = prev
155 prev = current
156 current = next
157
158 return prev
159
160 class HashTableEntry:
161 """
162 Linked List hash table key/value pair
163 """
164 def __init__(self, key, value):
165 self.key = key
166 self.value = value
167 self.next = None
168

```

```
169
170 # Hash table can't have fewer than this many slots
171 MIN_CAPACITY = 8
172
173 [
174 0["Lou", 41] -> ["Bob", 41] -> None,
175 1["Steve", 41] -> None,
176 2["Jen", 41] -> None,
177 3["Dave", 41] -> None,
178 4None,
179 5["Hector", 34]-> None,
180 6["Lisa", 41] -> None,
181 7None,
182 8None,
183 9None
184]
185 class HashTable:
186 """
187 A hash table that with `capacity` buckets
188 that accepts string keys
189 Implement this.
190 """
191
192 def __init__(self, capacity):
193 self.capacity = capacity # Number of buckets in the hash tabl
194 self.storage = [None] * capacity
195 self.item_count = 0
196
197
198 def get_num_slots(self):
199 """
200 Return the length of the list you're using to hold the hash
201 table data. (Not the number of items stored in the hash table,
202 but the number of slots in the main list.)
203 One of the tests relies on this.
204 Implement this.
205 """
206 # Your code here
207
208
209 def get_load_factor(self):
210 """
211 Return the load factor for this hash table.
212 Implement this.
213 """
214 return len(self.storage)
215
216
217 def djb2(self, key):
218 """
219 DJB2 hash, 32-bit
220 Implement this, and/or FNV-1.
221 """
```

```
222 str_key = str(key).encode()
223
224 hash = FNV_offset_basis_64
225
226 for b in str_key:
227 hash *= FNV_prime_64
228 hash ^= b
229 hash &= 0xffffffffffffffff # 64-bit hash
230
231 return hash
232
233
234 def hash_index(self, key):
235 """
236 Take an arbitrary key and return a valid integer index
237 between within the storage capacity of the hash table.
238 """
239 return self.djb2(key) % self.capacity
240
241 def put(self, key, value):
242 """
243 Store the value with the given key.
244 Hash collisions should be handled with Linked List Chaining.
245 Implement this.
246 """
247 index = self.hash_index(key)
248
249 current_entry = self.storage[index]
250
251 while current_entry is not None and current_entry.key != key:
252 current_entry = current_entry.next
253
254 if current_entry is not None:
255 current_entry.value = value
256 else:
257 new_entry = HashTableEntry(key, value)
258 new_entry.next = self.storage[index]
259 self.storage[index] = new_entry
260
261
262 def delete(self, key):
263 """
264 Remove the value stored with the given key.
265 Print a warning if the key is not found.
266 Implement this.
267 """
268 # Your code here
269
270
271 def get(self, key):
272 """
273 Retrieve the value stored with the given key.
274 Returns None if the key is not found.
```

```
275 Implement this.
276 """
277 # Your code here
```

---

# D2

---

## **Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a queue**

### **Overview**

A queue is a data structure that stores its items in a first-in, first-out (FIFO) order. That is precisely why it is called a queue. It functions just like a queue (or a line) would in everyday life. If you are the first to arrive at the check-in desk at a hotel, you will be the first to be served (and therefore, the first person to exit the queue). So, in other words, the items that are added to the queue first are the first items to be removed from the queue.

### **Follow Along**

#### **Time and Space Complexity**

##### **Enqueue**

To enqueue an item (add an item to the back of the queue) takes  $O(1)$  time.

##### **Dequeue**

To dequeue an item (remove an item from the front of the queue) takes  $O(1)$  time.

##### **Peek**

To peek at an item (inspect the item from the front of the queue without removing it) takes  $O(1)$  time.

##### **Space**

The space complexity of a queue is linear ( $O(n)$ ). Each item in the queue will take up space in memory.

## Strengths

The primary strength of a queue is that all of its operations are fast (take  $O(1)$  time).

## Weaknesses

There are no weaknesses in this data structure. The reason is that it is a very targeted data structure designed to do a few things well.

## When are queues useful?

Queues are useful data structures in any situation where you want to make sure things are processed in a FIFO order. Think of a web server. The server might be trying to service thousands of page requests per minute. It would make the most sense for the server to process and respond to the requests in the same order that they were received. That way, the first client to request a page is the first client to receive a response. Also, you'll learn soon enough about traversing hierarchical data structures. One of the ways you do that is called a breadth-first traversal. To conduct a breadth-first traversal, a queue can be used.

## Challenge

1. In your own words, explain the strengths of a queue data structure.
2. If a queue only allows operations at the ends (front and back), what other data structure would be a perfect one to build the queue?

## Additional Resources

- <https://www.geeksforgeeks.org/queue-data-structure/> (Links to an external site.)

---

## Objective 02 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a stack

## Overview

A stack data structure handles information in a last-in, first-out order. This means that the last item added to the storage will be the first item removed from the storage. A stack is like having a paper tray inbox on your desk. Anytime a person walks by and drops a piece of paper or a letter in your inbox, it will go on the top of your inbox. So, when you process your inbox, the first item you would remove from the top of the stack of papers would be the last item added to it.

## Follow Along

### Time and Space Complexity

#### Push

To push an item (add an item to the top of the stack) takes  $O(1)$  time.

#### Pop

To pop an item (remove an item from the top of the stack) takes  $O(1)$  time.

#### Peek

To peek at an item (inspect the item from the top of the stack without removing it) takes  $O(1)$  time.

#### Space

The space complexity of a stack is linear ( $O(n)$ ). Each item in the stack will take up space in memory.

#### Strengths

The primary strength of a stack is that all of its operations are fast (take  $O(1)$  time).

#### Weaknesses

There are no weaknesses in this data structure. The reason is that it is a very targeted data structure designed to do a few things well.

## When are stacks useful?

Stacks can be useful in any situation where you desire a LIFO order. One common use-case is for parsing strings. Let's say you wanted to parse a string to ensure that all the parentheses in your string are correctly nested. A stack could be useful for this. When you encounter an opening parenthesis, you add it to the stack. When you encounter a closing parenthesis, you remove the top opening parenthesis from the stack. After going through all the characters in the string, the stack should be empty. If it isn't or if you try to remove an item from an empty stack, you'll know that the parentheses were not correctly nested.

Additionally, function calls and execution contexts are managed on a call stack. When you call a function, it's added to the call stack. When it returns, it gets popped off of the stack. Last, an iterative depth-first-search can be done using a stack.

## Challenge

1. In your own words, explain the strengths of a stack data structure.
2. What two data structures would work well for implementing a stack?

## Additional Resources

- <https://www.geeksforgeeks.org/stack-data-structure/> (Links to an external site.)
- 

# Objective 03 - Implement a queue using a linked list

## Overview

To implement a queue, we need to maintain two pointers. One pointer will point at the front (the first item) of the queue, and another pointer will point at the rear (the last item) of the queue.

Additionally, we need to have two methods available: `enqueue()` and `dequeue()`.

`enqueue()` adds a new item after the rear. `dequeue()` removes the front node and resets the front pointer to the next node.

## Follow Along

We will use a `LinkedListNode` class for each of the items in the queue.

```
1 class LinkedListNode:
2 def __init__(self, data):
3 self.data = data
4 self.next = None
```

For our `Queue` class, we first need to define an `__init__` method. This method should initialize our instance variables `front` and `rear`.

```
1 class Queue:
2 def __init__(self):
3 self.front = None
4 self.rear = None
```

Next, we need to define our `enqueue` method:

```
1 class Queue:
2 def __init__(self):
3 self.front = None
4 self.rear = None
5 def enqueue(self, item):
6 new_node = LinkedListNode(item)
7 # check if queue is empty
8 if self.rear is None:
9 self.front = new_node
10 self.rear = new_node
11 else:
12 # add new node to rear
13 self.rear.next = new_node
14 # reassign rear to new node
15 self.rear = new_node
```

Now, we need to define our `dequeue` method:

```
1 class Queue:
2 def __init__(self):
```

```

3 self.front = None
4 self.rear = None
5 def enqueue(self, item):
6 new_node = LinkedListNode(item)
7 # check if queue is empty
8 if self.rear is None:
9 self.front = new_node
10 self.rear = new_node
11 else:
12 # add new node to rear
13 self.rear.next = new_node
14 # reassign rear to new node
15 self.rear = new_node
16 def dequeue(self):
17 # check if queue is empty
18 if self.front is not None:
19 # keep copy of old front
20 old_front = self.front
21 # set new front
22 self.front = old_front.next
23
24 # check if the queue is now empty
25 if self.front is None:
26 # make sure rear is also None
27 self.rear = None
28
29 return old_front

```

Now we have a `Queue` class that uses a singly-linked list as the underlying data structure.



[cs-unit-1-sprint-2-module-3-queue-with-linked-list-2](https://github.com/bgoonz/cs-unit-1-sprint-2-module-3-queue-with-linked-list-2)

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-queue-with-linked-list-2#main.py>

## Queue – Linked List Implementation

- Difficulty Level : Easy
- Last Updated : 06 Aug, 2021

In the previous post, we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a Queue data structure, we maintain two pointers, front and rear. The front points the first item of queue and rear points to last item.

**enQueue()** This operation adds a new node after rear and moves rear to the next node.

**deQueue()** This operation removes the front node and moves front to the next node.

Recommended: Please solve it on “**PRACTICE**” first, before moving on to the solution.

- C++
- C
- Java
- Python3
- C#
- Javascript

```
1 # Python3 program to demonstrate linked list
2 # based implementation of queue
3
4 # A linked list (LL) node
5 # to store a queue entry
6 class Node:
7
8 def __init__(self, data):
9 self.data = data
10 self.next = None
11
12 # A class to represent a queue
13
14 # The queue, front stores the front node
15 # of LL and rear stores the last node of LL
16 class Queue:
17
18 def __init__(self):
19 self.front = self.rear = None
20
21 def isEmpty(self):
22 return self.front == None
23
24 # Method to add an item to the queue
25 def EnQueue(self, item):
26 temp = Node(item)
27
```

```
28 if self.rear == None:
29 self.front = self.rear = temp
30 return
31 self.rear.next = temp
32 self.rear = temp
33
34 # Method to remove an item from queue
35 def DeQueue(self):
36
37 if self.isEmpty():
38 return
39 temp = self.front
40 self.front = temp.next
41
42 if(self.front == None):
43 self.rear = None
44
45 # Driver Code
46 if __name__ == '__main__':
47 q = Queue()
48 q.Enqueue(10)
49 q.Enqueue(20)
50 q.DeQueue()
51 q.DeQueue()
52 q.Enqueue(30)
53 q.Enqueue(40)
54 q.Enqueue(50)
55 q.DeQueue()
56 print("Queue Front " + str(q.front.data))
57 print("Queue Rear " + str(q.rear.data))
```

### Output:

```
1 Queue Front : 40
2 Queue Rear : 50
```

### Output:

```
1 Queue Front : 40
2 Queue Rear : 50
```

**Time Complexity:** Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

## D4



traversals.ipynb

<https://gist.github.com/bgoonz/ff875d63c2b0b6763531ccbf1f3ef637#file-traversals-ipynb>

WK-20

# D3

---

## **Objective 02 - Represent a graph as an adjacency list and an adjacency matrix and compare and contrast the respective representations**

### **Overview**

#### **Graph Representations**

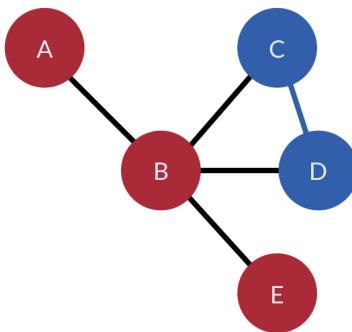
Two common ways to represent graphs in code are **adjacency lists** and **adjacency matrices**. Both of these options have strengths and weaknesses. When deciding on a graph implementation, it's essential to understand what type of data you will store and what operations you need to run on the graph.

Below is an example of how we would represent a graph with an adjacency matrix and an adjacency list. Notice how we represent the relationship between verts C and D when using each type.

## Adjacency Matrix

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	1
C	0	1	0	1	0
D	0	1	1	0	0
E	0	1	0	0	0

## Graph



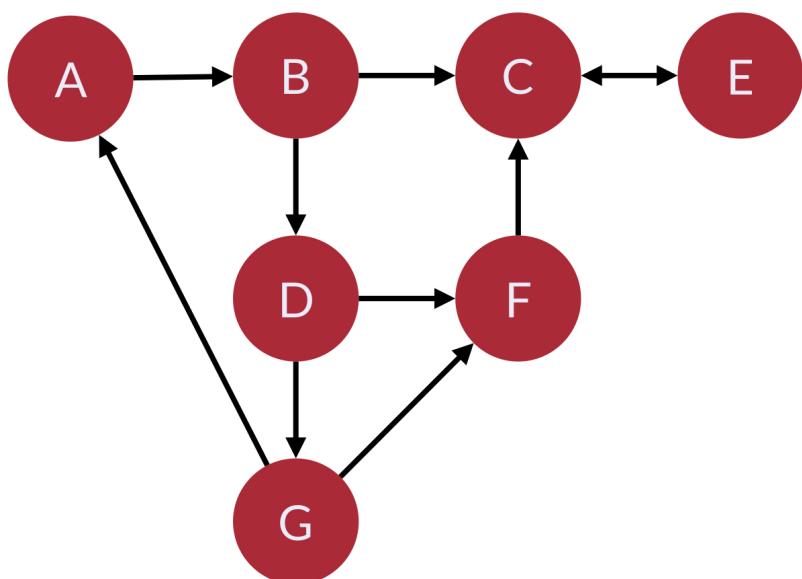
## Adjacency List

```
{
 A: = > B
 B: = > A, C, D, E
 C: = > B,
 D: = > B, C
 E: = > B
}
```

<https://camo.githubusercontent.com/ff694105bfdaea68ee3a73c75cf604ac8f020e1c/68747470733a2f2f692e696d6775722e636f6d2f7369476d7138582e6a7067>

## Adjacency List

In an adjacency list, the graph stores a list of vertices. For each vertex, it holds a list of each connected vertex.



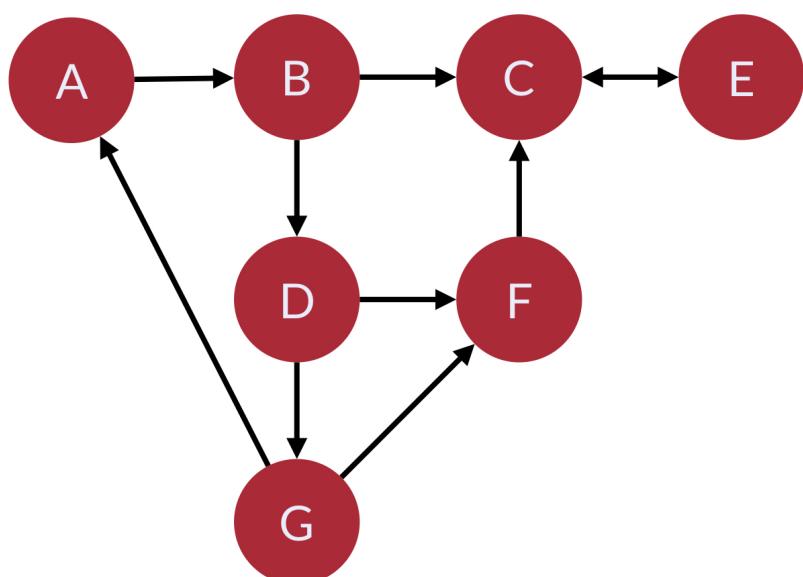
<https://camo.githubusercontent.com/0e81024228bd0b1dd29f33c47b0896b7a978e911/6874740733a2f2f692e696d6775722e636f6d2f476953746d4e682e6a7067>

Below is a representation of the graph above in Python:

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B"},
5 "B": {"C", "D"},
6 "C": {"E"},
7 "D": {"F", "G"},
8 "E": {"C"},
9 "F": {"C"},
10 "G": {"A", "F"}
11 }
12 }
```

Notice that this adjacency *list* doesn't use any lists. The `vertices` collection is a `dictionary` which lets us access each collection of edges in  $O(1)$  constant time. Because a `set` contains the edges, we can check for edges in  $O(1)$  constant time.

## Adjacency Matrix



<https://camo.githubusercontent.com/0e81024228bd0b1dd29f33c47b0896b7a978e911/6874>

7470733a2f2f692e696d6775722e636f6d2f476953746d4e682e6a7067

Here is the representation of the graph above in an adjacency matrix:

```
1 class Graph:
2 def __init__(self):
3 self.edges = [[0,1,0,0,0,0,0,0],
4 [0,0,1,1,0,0,0],
5 [0,0,0,0,1,0,0],
6 [0,0,0,0,0,1,1],
7 [0,0,1,0,0,0,0],
8 [0,0,1,0,0,0,0],
9 [1,0,0,0,0,1,0]]
```

We represent this matrix as a two-dimensional array—a list of lists. With this implementation, we get the benefit of built-in edge weights. `0` denotes no relationship, but any other value represents an edge label or edge weight. The drawback is that we do not have a built-in association between the vertex values and their index.

In practice, implementing both the adjacency list and adjacency matrix would contain more information by including `Vertex` and `Edge` classes.

## Tradeoffs

Adjacency matrices and adjacency lists have strengths and weaknesses. Let's explore their tradeoffs by comparing their attributes and the efficiency of operations.

In all the following examples, we are using the following shorthand to denote the graph's properties:

Shorthand	Property
V	Total number of vertices in the graph
E	Total number of edges in the graph
e	Average number of edges per vertex

## Space Complexity

## Adjacency Matrix

*Complexity:*  $O(V^2)$  space

Consider a dense graph where each vertex points to each other vertex. Here, the total number of edges will approach  $V^2$ . This fact means that regardless of whether you choose an adjacency list or an adjacency matrix, both will have a comparable space complexity. However, dictionaries and sets are less space-efficient than lists. So, for dense graphs (graphs with a high average number of edges per vertex), the adjacency matrix is more efficient because it uses lists instead of dictionaries and sets.

## Adjacency List

*Complexity:*  $O(V+E)$  space

Consider a sparse graph with 100 vertices and only one edge. An adjacency list would have to store all 100 vertices but only needs to keep track of that single edge. The adjacency matrix would need to store  $100 \times 100 = 10,000$  connections, even though all but one would be 0.

*Takeaway:* *The worst-case storage of an adjacency list occurs when the graph is dense. The matrix and list representation have the same complexity ( $O(V^2)$ ). However, for the general case, the list representation is usually more desirable. Also, since finding a vertex's neighbors is a common task, and adjacency lists make this operation more straightforward, it is most common to use adjacency lists to represent graphs.*

## Add Vertex

### Adjacency Matrix

*Complexity:*  $O(V)$  time

For an adjacency matrix, we would need to add a new value to the end of each existing row and add a new row.

```
1 for v in self.edges:
2 self.edges[v].append(0)
3 v.append([0] * len(self.edges + 1))
```

```
for v in self.edges: self.edges[v].append(0)
v.append([0] * len(self.edges + 1))
```

Remember that with Python lists, appending to the end of a list is  $O(1)$  because of over-allocation of memory but can be  $O(n)$  when the over-allocated memory fills up. When this occurs, adding the vertex can be  $O(V^2)$ .

## Adjacency List

*Complexity:*  $O(1)$  time

Adding a vertex is simple in an adjacency list:

```
self.vertices["H"] = set()
```

Adding a new key to a dictionary is a constant-time operation.

*Takeaway:* Adding vertices is very inefficient for adjacency matrices but very efficient for adjacency lists.

## Remove Vertex

### Adjacency Matrix

*Complexity:*  $O(V^2)$

Removing vertices is inefficient in both representations. In an adjacency matrix, we need to remove the removed vertex's row and then remove that column from each row. Removing an element from a list requires moving everything after that element over by one slot, which takes an average of  $V/2$  operations. Since we need to do that for every single row in our matrix, that results in  $V^2$  time complexity. We need to reduce each vertex index after our removed index by one as well, which doesn't add to our quadratic time complexity but adds extra operations.

### Adjacency List

*Complexity:*  $O(V)$

We need to visit each vertex for an adjacency list and remove all edges pointing to our removed vertex. Removing elements from sets and dictionaries is an  $O(1)$  operation, resulting in an overall  $O(v)$  time complexity.

*Takeaway: Removing vertices is inefficient in both adjacency matrices and lists but more efficient in lists.*

## Add Edge

### Adjacency Matrix

*Complexity:*  $O(1)$

Adding an edge in an adjacency matrix is simple:

```
self.edges[v1][v2] = 1
```

### Adjacency List

*Complexity:*  $O(1)$

Adding an edge in an adjacency list is simple:

```
self.vertices[v1].add(v2)
```

Both are constant-time operations.

*Takeaway: Adding edges to both adjacency matrices and lists is very efficient.*

## Remove Edge

### Adjacency Matrix

*Complexity:*  $O(1)$

Removing an edge from an adjacency matrix is simple:

```
self.edges[v1][v2] = 0
```

## Adjacency List

*Complexity:*  $O(1)$

Removing an edge from an adjacency list is simple:

```
self.vertices[v1].remove(v2)
```

Both are constant-time operations.

*Takeaway:* *Removing edges from both adjacency matrices and lists is very efficient.*

## Find Edge

### Adjacency Matrix

*Complexity:*  $O(1)$

Finding an edge in an adjacency matrix is simple:

```
return self.edges[v1][v2] > 0
```

### Adjacency List

*Complexity:*  $O(1)$

Finding an edge in an adjacency list is simple:

```
return v2 in self.vertices[v1]
```

Both are constant-time operations.

*Takeaway: Finding edges in both adjacency matrices and lists is very efficient.*

## Get All Edges from Vertex

You can use several commands if you want to know all the edges originating from a particular vertex.

### Adjacency Matrix

*Complexity:*  $O(V)$

In an adjacency matrix, this is complicated. You would need to iterate through the entire row and populate a list based on the results:

```
1 v_edges = []
2 for v2 in self.edges[v]:
3 if self.edges[v][v2] > 0:
4 v_edges.append(v2)
5 return v_edges
```

### Adjacency List

*Complexity:*  $O(1)$

With an adjacency list, this is as simple as returning the value from the vertex dictionary:

```
return self.vertex[v]
```

*Takeaway: Fetching all edges is less efficient in an adjacency matrix than an adjacency list.*

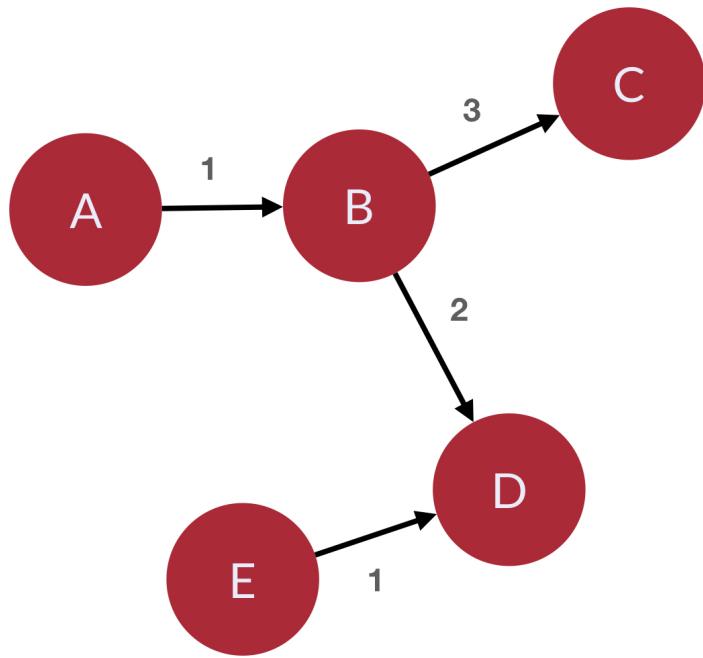
## Summary

Let's summarize all this complexity information in a table:

type	Space	Add Vert	Remove Vert	Add Edge	Remove Edge	Find Edge	Get All Edges
Matrix	$O(V^2)$	$O(V)$	$O(V^2)$	$O(1)$	$O(1)$	$O(1)$	$O(V)$
List	$O(V+E)$	$O(1)$	$O(V)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

In most practical use-cases, an adjacency list will be a better choice for representing a graph. However, it is also crucial that you be familiar with the matrix representation. Why? Because there are some dense graphs or weighted graphs that could have better space efficiency when represented by a matrix.

## Follow Along



<https://camo.githubusercontent.com/335012587396b095af8f6a8f28e2d2aedb3d84d0/68747470733a2f2f692e696d6775722e636f6d2f796931503441462e6a7067>

Together, we will now use the graph shown in the picture above and represent it in both an adjacency list and an adjacency matrix.

## Adjacency List

First, the adjacency list:

```
1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B": 1},
5 "B": {"C": 3, "D": 2},
6 "C": {},
7 "D": {},
8 "E": {"D": 1}
9 }
10 }
```

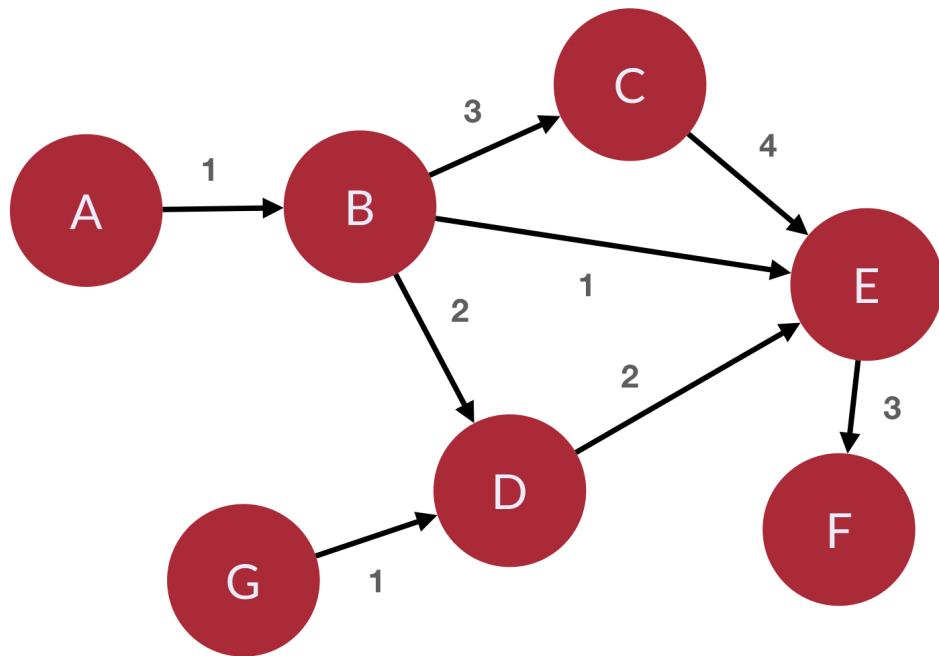
The difference between this implementation and the previous adjacency list is that this representation allows our edges to have weights.

## Adjacency Matrix

Now, we need to implement an adjacency matrix. Remember, that one benefit of the matrix is how easy it is to represent edge weights:

```
1 class Graph:
2 def __init__(self):
3 self.edges = [[0,1,0,0,0],
4 [0,0,3,2,0],
5 [0,0,0,0,0],
6 [0,0,0,0,0],
7 [0,0,0,1,0]]
```

## Challenge



<https://camo.githubusercontent.com/b6251eb484344b565ae2753682c645f85283ab28/68747470733a2f2f692e696d6775722e636f6d2f634a366c656b4d2e6a7067>

1. Using the graph shown in the picture above, write python code to represent the graph in an adjacency list.
2. Using the same graph you used for the first exercise, write python code to represent the graph in an adjacency matrix.
3. Write a paragraph that compares and contrasts the efficiency of your different representations.

## Additional Resources

- <https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs> (Links to an external site.)

Tags

---

**Previous** **Next**

# D2

## Objective 01 - Represent a breadth-first-search of a graph in pseudo-code and recall typical applications for its use

### Overview

One method we can use when searching a graph is a **breadth-first search** (BFS). This sorting algorithm explores the graph outward in rings of increasing distance from the starting vertex.

The algorithm never attempts to explore a vert it has already explored or is currently exploring.

For example, when starting from the upper left, the numbers on this graph show a vertex



We followed the edges represented with thicker black arrows. We did not follow the edges represented with thinner grey arrows because we already visited their destination nodes.

The exact order will vary depending on which branches get taken first and which vertex is the starting vertex.

*Note: it's essential to know the distinction between a breadth-first search and a breadth-first traversal. A breadth-first traversal is when you visit each vertex in the breadth-first order and do something during the traversal. A breadth-first search is when you search through vertexes in the breadth-first order until you find the target vertex. A breadth-first search usually returns the shortest path from the starting vertex to the target vertex once the target is found.*

### Applications of BFS

- Pathfinding, Routing
- Find neighbor nodes in a P2P network like BitTorrent
- Web crawlers

- Finding people  $n$  connections away on a social network
- Find neighboring locations on the graph
- Broadcasting in a network
- Cycle detection in a graph
- Finding Connected Components (Links to an external site.)
- Solving several theoretical graph problems

## Coloring Vertices

As we explore the graph, it is useful to color verts as we arrive at them and as we leave them behind as "already searched".

Unvisited verts are white, verts whose neighbors are being explored are gray, and verts with no unexplored neighbors are black.

## Keeping Track of What We Need to Explore

In a BFS, it's useful to track which nodes we still need to explore. For example, in the diagram above, when we get to node 2, we know that we also need to explore nodes 3 and 4.

We can track that by adding neighbors to a *queue* (which remember is first in, first out), and then explore the verts in the queue one by one.

## Follow Along

### Pseudo-code for BFS

Let's explore some pseudo-code that shows a basic implementation of a breadth-first-search of a graph. Make sure you can read the pseudo-code and understand what each line is doing before moving on.

```

1 BFS(graph, startVert):
2 for v of graph.vertices:
3 v.color = white
4
5 startVert.color = gray
6 queue.enqueue(startVert)
7
8 while !queue.isEmpty():
9 u = queue[0] // Peek at head of the queue, but do not dequeue!

```

```

10
11 for v in u.neighbors:
12 if v.color == white:
13 v.color = gray
14 queue.enqueue(v)
15
16 queue.dequeue()
17 u.color = black

```

You can see that we start with a graph and the vertex we will start on. The very first thing we do is go through each of the vertices in the graph and mark them with the color white. At the outset, we mark all the verts as unvisited.

Next, we mark the starting vert as gray. We are exploring the starting verts' neighbors. We also enqueue the starting vert, which means it will be the first vert we look at once we enter the while loop.

The condition we check at the outset of each while loop is if the queue is **not** empty. If it is not empty, we peek at the first item in the queue by storing it in a variable.

Then, we loop through each of that vert's neighbors and:

- We check if it is unvisited (the color white).
- If it is unvisited, we mark it as gray (meaning we will explore its neighbors).
- We enqueue the vert.

Next, we dequeue the current vert we've been exploring and mark that vert as black (marking it as visited).

We continue with this process until we have explored all the verts in the graph.

## Challenge

On your own, complete the following tasks:

1. Please spend a few minutes researching to find a unique use-case of a breadth-first-search that we did not mention in the list above.
2. Using the graph represented below, draw a picture of the graph and label each of the verts to show the correct vertex visitation order for a breadth-first-search starting with vertex "I".

```

1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B", "C", "D"},
5 "B": {},
6 "C": {"E", "F"},
7 "D": {"G"},
8 "E": {"G"},
9 "F": {"J"},
10 "G": {},
11 "H": {"C", "J", "K"},
12 "I": {"D", "E", "H"},
13 "J": {"L"},
14 "K": {"C"},
15 "L": {"M"},
16 "M": {},
17 "N": {"H", "K", "M"}
18 }

```

3. Besides marking verts with colors as in the pseudo-code example above, how else could you track the verts we have already visited?

## Additional Resources

- <https://brilliant.org/wiki/breadth-first-search-bfs/>

Breadth-First Search (BFS) | Brilliant Math & Science Wiki Breadth-first search (BFS) is an important graph search algorithm that is used to solve many problems including finding the shortest path in a graph and solving puzzle games (such as Rubik's Cubes). Many problems in computer science can be thought of in terms of graphs.<https://brilliant.org/wiki/breadth-first-search-bfs/>




---

## Objective 02 - Represent a depth-first-search of a graph in

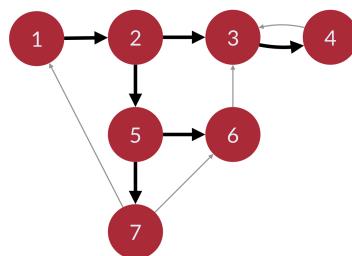
# pseudo-code and recall typical applications for its use

## Overview

Another method we can use when searching a graph is a **depth-first search** (DFS). This searching algorithm "dives" "down" the graph as far as it can before backtracking and exploring another branch.

The algorithm never attempts to explore a vert it has already explored or is exploring.

For example, when starting from the upper left, the numbers on this graph show a vertex



visitation order in a DFS:

We followed the edges represented with thicker black arrows. We did not follow the edges represented with thinner grey arrows because we already visited their destination nodes.

The exact order will vary depending on which branches get taken first and which vertex is the starting vertex.

## Applications of DFS

DFS is often the preferred method for exploring a graph *if we want to ensure we visit every node in the graph*. For example, let's say that we have a graph representing all the friendships in the entire world. We want to find a path between two known people, `Andy` and `Sarah`. If we used a depth-first search in this scenario, we could end up exceptionally far away from `Andy` while still not finding a path to `Sarah`. Using a DFS, we will eventually find the path, but it won't find the shortest route, and it will also likely take a long time.

So, this is an example of where a DFS *would not work well*. What about a genuine use case for DFS. Here are a few examples:

- Finding Minimum Spanning Trees (Links to an external site.) of weighted graphs
- Pathfinding

- Detecting cycles in graphs
- Topological sorting (Links to an external site.), useful for scheduling sequences of dependent jobs
- Solving and generating mazes

## Coloring Vertices

Again, as we explore the graph, it is useful to color verts as we arrive at them and as we leave them behind as "already searched".

Unvisited verts are white, verts whose neighbors are being explored are gray, and verts with no unexplored neighbors are black.

## Recursion

Since DFS will pursue leads in the graph as far as it can, and then "back up" to an earlier branch point to explore that way, recursion is an excellent approach to help "remember" where we left off.

Looking at it with pseudo-code to make the recursion more clear:

```

1 explore(graph) {
2 visit(this_vert);
3 explore(remaining_graph);
4 }

```

## Follow Along

### Pseudo-code for DFS

Let's explore some pseudo-code that shows a basic implementation of a depth-first-search of a graph. Make sure you can read the pseudo-code and understand what each line is doing before moving on.

```

1 DFS(graph):
2 for v of graph.verts:
3 v.color = white
4 v.parent = null

```

```

5
6 for v of graph.verts:
7 if v.color == white:
8 DFS_visit(v)
9
10 DFS_visit(v):
11 v.color = gray
12
13 for neighbor of v.adjacent_nodes:
14 if neighbor.color == white:
15 neighbor.parent = v
16 DFS_visit(neighbor)
17
18 v.color = black

```

You can see that we have two functions in our pseudo-code above. The first function, `DFS()` takes the graph as a parameter and marks all the verts as unvisited (white). It also sets the parent of each vert to `null`. The next loop in this function visits each vert in the graph, and if it is unvisited, it passes that vert into our second function `DFS_visit()`.

`DFS_visit()` starts by marking the vert as gray (in the process of being explored). Then, it loops through all of its unvisited neighbors. In that loop, it sets the parent and then makes a recursive call to the `DFS_visit()`. Once it's done exploring all the neighbors, it marks the vert as black (visited).

## Challenge

On your own, complete the following tasks:

1. Please spend a few minutes researching to find a unique use-case of a depth-first search that we did not mention in the list above.
2. Using the graph represented below, draw a picture of the graph and label each of the verts to show the correct vertex visitation order for a depth-first-search starting with vertex "I".

```

1 class Graph:
2 def __init__(self):
3 self.vertices = {
4 "A": {"B", "C", "D"},
5 "B": {},
6 "C": {"E", "F"},
7 "D": {"G"},
8 "E": {"G"},
```

```
9 "F": {"J"},
10 "G": {},
11 "H": {"C", "J", "K"},
12 "I": {"D", "E", "H"},
13 "J": {"L"},
14 "K": {"C"},
15 "L": {"M"},
16 "M": {},
17 "N": {"H", "K", "M"}
18 }
```

## Additional Resources

- <https://brilliant.org/wiki/depth-first-search-dfs/>
- 

# Objective 03 - Implement a breadth-first search on a graph

## Overview

Now that we've looked at and understand the basics of a breadth-first search (BFS) on a graph, let's implement a BFS algorithm.

## Follow Along

Before defining our breadth-first search method, review our `Vertex` and `Graph` classes that we defined previously.

```
1 class Vertex:
2 def __init__(self, value):
3 self.value = value
4 self.connections = {}
5
6 def __str__(self):
7 return str(self.value) + ' connections: ' + str([x.value for x in self.connections])
8
9 def add_connection(self, vert, weight = 0):
10 self.connections[vert] = weight
11
12 def get_connections(self):
```

```

13 return self.connections.keys()
14
15 def get_value(self):
16 return self.value
17
18 def get_weight(self, vert):
19 return self.connections[vert]

```

```

1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0
5
6 def __contains__(self, vert):
7 return vert in self.vertices
8
9 def __iter__(self):
10 return iter(self.vertices.values())
11
12 def add_vertex(self, value):
13 self.count += 1
14 new_vert = Vertex(value)
15 self.vertices[value] = new_vert
16 return new_vert
17
18 def add_edge(self, v1, v2, weight = 0):
19 if v1 not in self.vertices:
20 self.add_vertex(v1)
21 if v2 not in self.vertices:
22 self.add_vertex(v2)
23 self.vertices[v1].add_connection(self.vertices[v2], weight)
24
25 def get_vertices(self):
26 return self.vertices.keys()

```

Now, we will add a `breadth_first_search` method to our `Graph` class. One of the most common and simplest ways to implement a BFS is to use a queue to keep track of unvisited nodes and a set to keep track of visited nodes. Let's start by defining the start of our function with these structures:

```

1 class Graph:
2 def __init__(self):
3 self.vertices = {}
4 self.count = 0

```

```

5 def __contains__(self, vert):
6 return vert in self.vertices
7
8 def __iter__(self):
9 return iter(self.vertices.values())
10
11
12 def add_vertex(self, value):
13 self.count += 1
14 new_vert = Vertex(value)
15 self.vertices[value] = new_vert
16 return new_vert
17
18 def add_edge(self, v1, v2, weight = 0):
19 if v1 not in self.vertices:
20 self.add_vertex(v1)
21 if v2 not in self.vertices:
22 self.add_vertex(v2)
23 self.vertices[v1].add_connection(self.vertices[v2], weight)
24
25 def get_vertices(self):
26 return self.vertices.keys()
27
28 def breadth_first_search(self, starting_vert):
29 to_visit = Queue()
30 visited = set()
31 to_visit.enqueue(starting_vert)
32 visited.add(starting_vert)
33 while to_visit.size() > 0:
34 current_vert = to_visit.dequeue()
35 for next_vert in current_vert.get_connections():
36 if next_vert not in visited:
37 visited.add(next_vert)
38 to_visit.enqueue(next_vert)

```

## Challenge

1. What is time complexity in Big O notation of a breadth-first search on a graph with  $v$  vertices and  $E$  edges?
2. Which method will find the ***shortest*** path between a starting point and any other reachable node? A breadth-first search or a depth-first search?

## Additional Resources

- <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

# D4

---

## **Objective 01 - Recall the different traversal types for a binary tree and implement a function to complete the traversal for each type**

### **Overview**

There is only one way to traverse linear data structures like arrays, linked lists, queues, and stacks. The linear nature of the structure itself forces a particular type of traversal.

However, with hierarchical structures like trees, there are multiple ways that you can traverse the stored data. There are two primary categories for tree traversals:

1. Depth-First
2. Breadth-First

Furthermore, there are three different types of depth-first traversals:

1. Inorder
2. Preorder
3. Postorder

Let's dive deeper into each of the traversal types.

### **Follow Along**

#### **Depth-First Inorder Traversal**

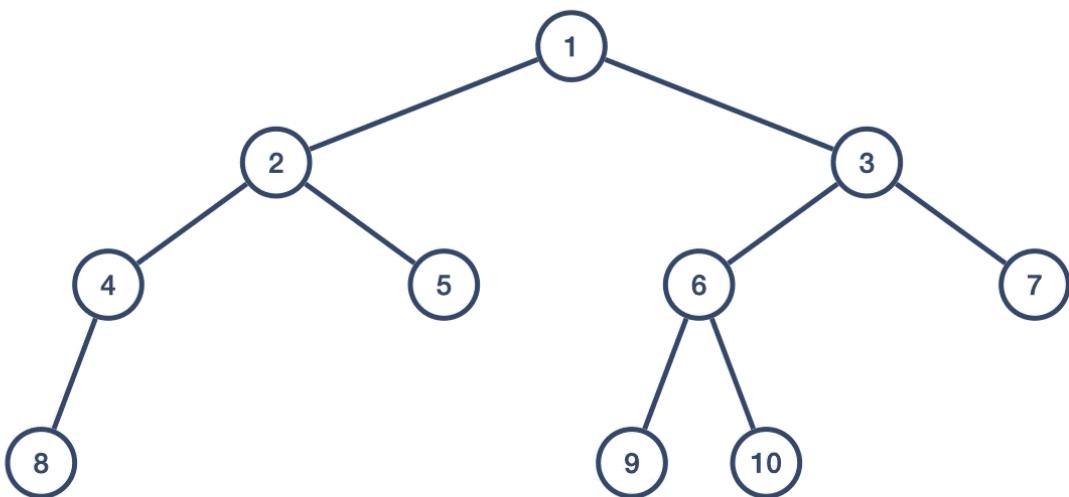
Let's first look at an inorder depth-first traversal of a binary tree. In this traversal, we start at the tree's root node and complete the following steps recursively:

1. Go to the left subtree

2. Visit node
3. Go to the right subtree

Notice that we don't actually "visit" a node until we've already gone to the left subtree. In the animation below, the "going" is denoted by changing the color to a light grey. The actual visiting is represented when it turns red. The base cases in the recursion are when there is no left or right subtree to visit.

## depth-first inorder traversal



[https://tk-assets.lambdaschool.com/4b1680ed-3b4b-4fcf-ba97-bbfe54f5d066\\_depth-first-inorder-traversal.gif](https://tk-assets.lambdaschool.com/4b1680ed-3b4b-4fcf-ba97-bbfe54f5d066_depth-first-inorder-traversal.gif)

Here is one possible way to code a depth-first inorder traversal in Python:

```
1 class TreeNode:
2 def __init__(self, val=0, left=None, right=None):
3 self.val = val
4 self.left = left
5 self.right = right
6
7 def helper(root, res):
8 if root is None:
9 return
10 helper(root.left, res)
11 res.append(root.val)
12 helper(root.right, res)
13
```

```
14 def inorder_traversal(root):
15 result = []
16 helper(root, result)
17 return result
```

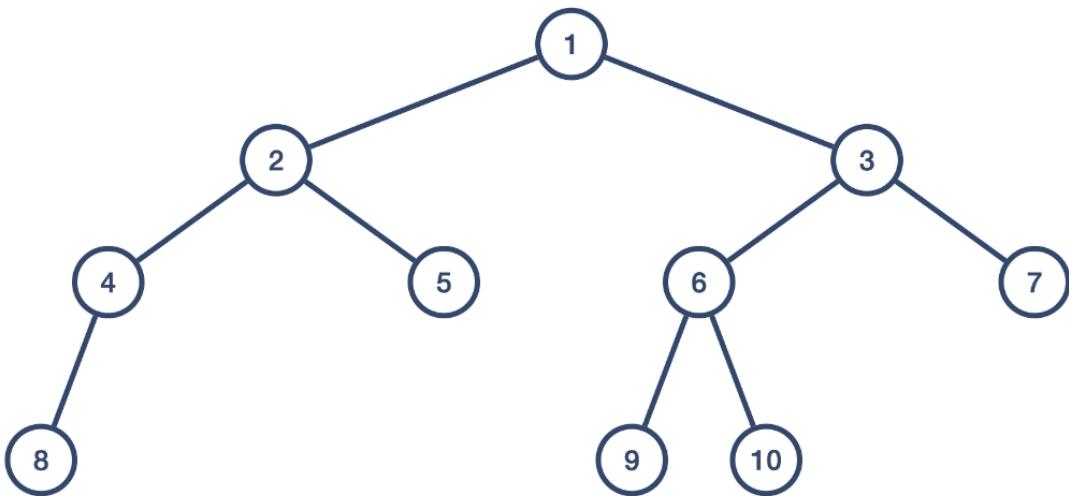
## Depth-First Preorder Traversal

This traversal type is very similar to an inorder traversal except that the three steps' order is slightly different. Notice that in this traversal, we "visit" the node (denoted in the visualization below by the node turning red) before we recurse to the left subtree (we represent the recursive call by turning the node grey in the visualization below). In the inorder traversal above, we recursed to the left subtree before visiting the node.

1. Visit the node
2. Go to the left subtree
3. Go to the right subtree

Below is the visualization for how this type of traversal would look.

### depth-first preorder traversal



[https://tk-assets.lambdaschool.com/c44685b7-b6f7-4214-ba85-226ca56e8042\\_depth-first-preorder-traversal.gif](https://tk-assets.lambdaschool.com/c44685b7-b6f7-4214-ba85-226ca56e8042_depth-first-preorder-traversal.gif)

Here is one possible way to code a depth-first preorder traversal in Python:

```

1 class TreeNode:
2 def __init__(self, val=0, left=None, right=None):
3 self.val = val
4 self.left = left
5 self.right = right
6
7 def helper(root, res):
8 if root is None:
9 return
10 res.append(root.val)
11 helper(root.left, res)
12 helper(root.right, res)
13
14 def preorder_traversal(root):
15 result = []
16 helper(root, result)
17 return result

```

Notice that the only difference between the code above for preorder traversal and the example for inorder traversal is that in the preorder traversal code, we append the node's value to the result before we recurse to the left.

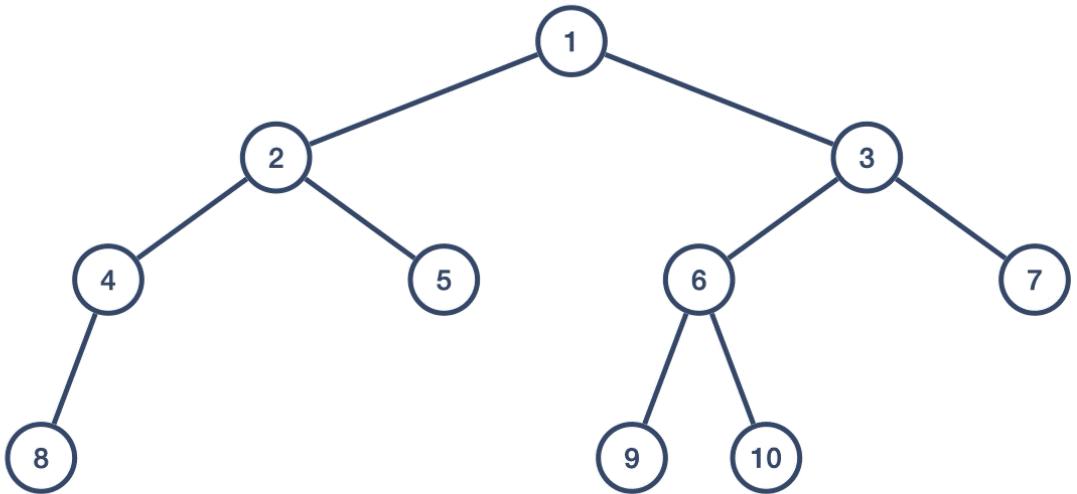
## Depth-First Postorder Traversal

This traversal type is very similar to our other traversals except that the three steps' order is slightly different. Notice that in this traversal, we "visit" the node (denoted in the visualization below by the node turning red) after we recurse to the left subtree (we represent the recursive call by turning the node grey in the visualization below) and the right subtree.

1. Go to the left subtree
2. Go to the right subtree
3. Visit node

Below is the visualization for how this would type of traversal would look.

## depth-first postorder traversal



[https://tk-assets.lambdaschool.com/41bc2877-94d4-4103-885b-c396bec4832a\\_depth-first-postorder-traversal.gif](https://tk-assets.lambdaschool.com/41bc2877-94d4-4103-885b-c396bec4832a_depth-first-postorder-traversal.gif)

Here is one possible way to code a depth-first postorder traversal in Python:

```
1 class TreeNode:
2 def __init__(self, val=0, left=None, right=None):
3 self.val = val
4 self.left = left
5 self.right = right
6
7 def helper(root, res):
8 if root is None:
9 return
10 helper(root.left, res)
11 helper(root.right, res)
12 res.append(root.val)
13
14 def postorder_traversal(self):
15 result = []
16 self.helper(self, result)
17 return result
```

Notice that the only difference between the code above for postorder traversal and the other examples is that in this version, we append the node's value to the result only after we've already recursed to the left and right subtrees.

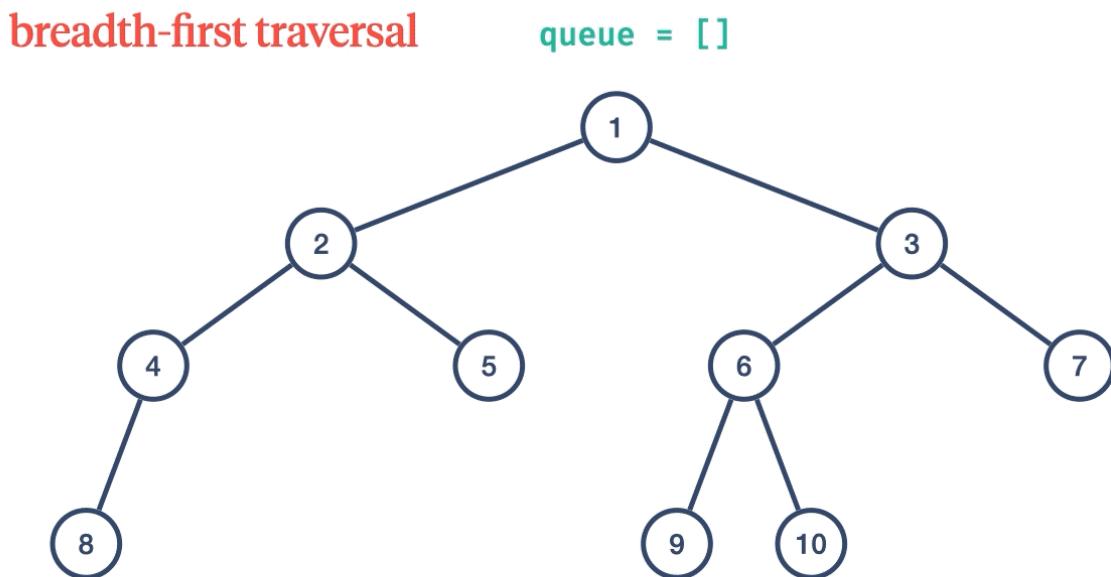
## Breadth-First (Level Order) Traversal

In a breadth-first traversal, we visit all the nodes at the same level (same distance from the root node) before going on to the next level.

A breadth-first traversal and a level order traversal are the same things. However, a breadth-first traversal can be done on any hierarchical data structure like trees and graphs. But, a level order traversal refers only to the traversal of a tree. Graphs do not have levels like trees do, so that term would not make sense.

A breadth-first traversal is a little different than the depth-first traversals we've gone over. We cannot merely use the recursive call stack to keep track of where we are in the tree. Instead, we must use a queue to keep track of what nodes we should visit. Remember that a queue data structure follows a first-in-first-out (FIFO) access order.

Below is a visualization for a breadth-first traversal.



[https://tk-assets.lambdaschool.com/671a11b7-acee-4b16-9452-d42f3b69a24e\\_breadth-first-traversal.gif](https://tk-assets.lambdaschool.com/671a11b7-acee-4b16-9452-d42f3b69a24e_breadth-first-traversal.gif)

Here is one way that you could code a breadth-first (level order) traversal in Python:

```
1 class TreeNode:
```

```
2 def __init__(self, val=0, left=None, right=None):
3 self.val = val
4 self.left = left
5 self.right = right
6
7 def breadth_first_traversal(root):
8 if root is None:
9 return []
10
11 result = []
12 queue = []
13 queue.append(root)
14
15 while len(queue) != 0:
16 node = queue.pop(0)
17 result.append(node.val)
18
19 if node.left is not None:
20 queue.append(node.left)
21
22 if node.right is not None:
23 queue.append(node.right)
24
25 return result
```

## Challenge

1. What data structure could you use to write an *iterative* depth-first traversal method?
2. In your own words, explain how a depth-first traversal and a breadth-first traversal are different?

## Additional Resources

- <https://www.geeksforgeeks.org/dfs-traversal-of-a-tree-using-recursion/> (Links to an external site.)
- <https://www.geeksforgeeks.org/level-order-tree-traversal/>

**python-language**

# General Notes On Graphs

## General Notes On Graphs

### Graphs

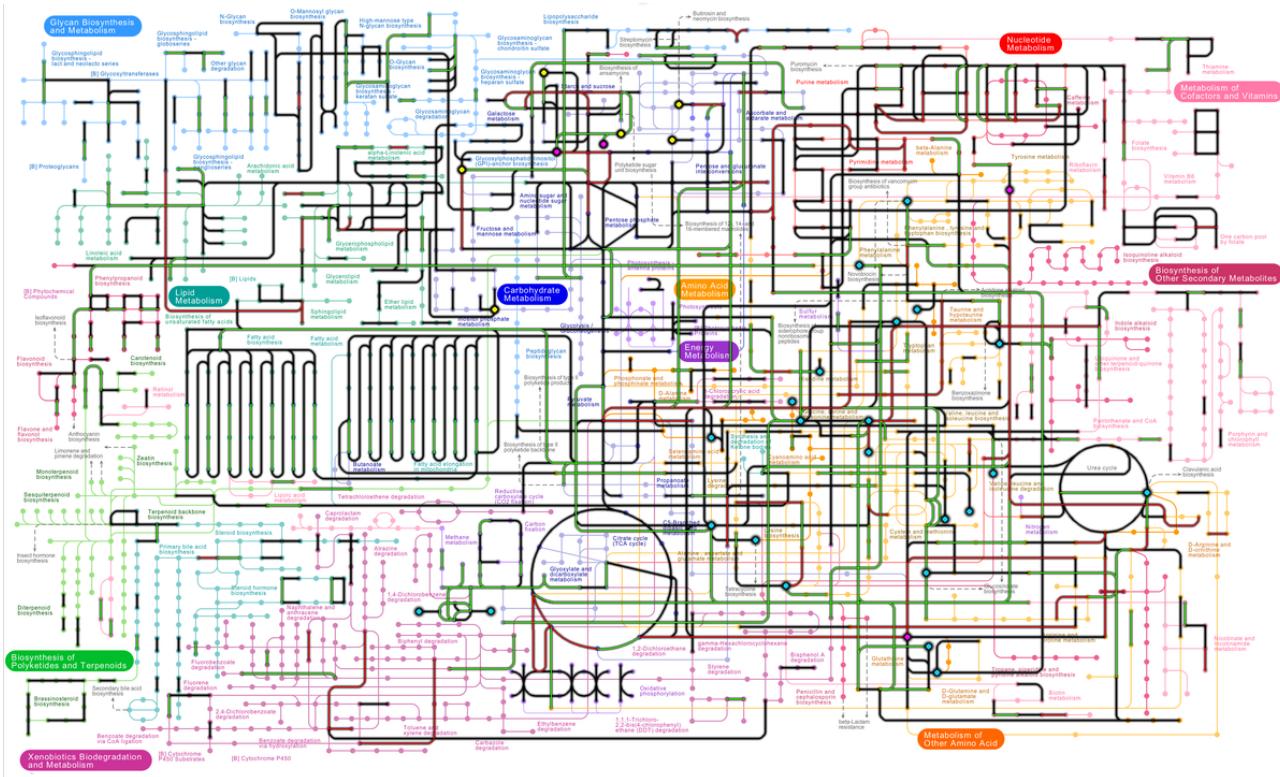
### Quizzes

- Graph fundamentals
- Breadth-first Search
- Depth-first Search

### Relevant For...

- Computer Science>Graphs
- Quantitative Finance>Computer Science Concepts

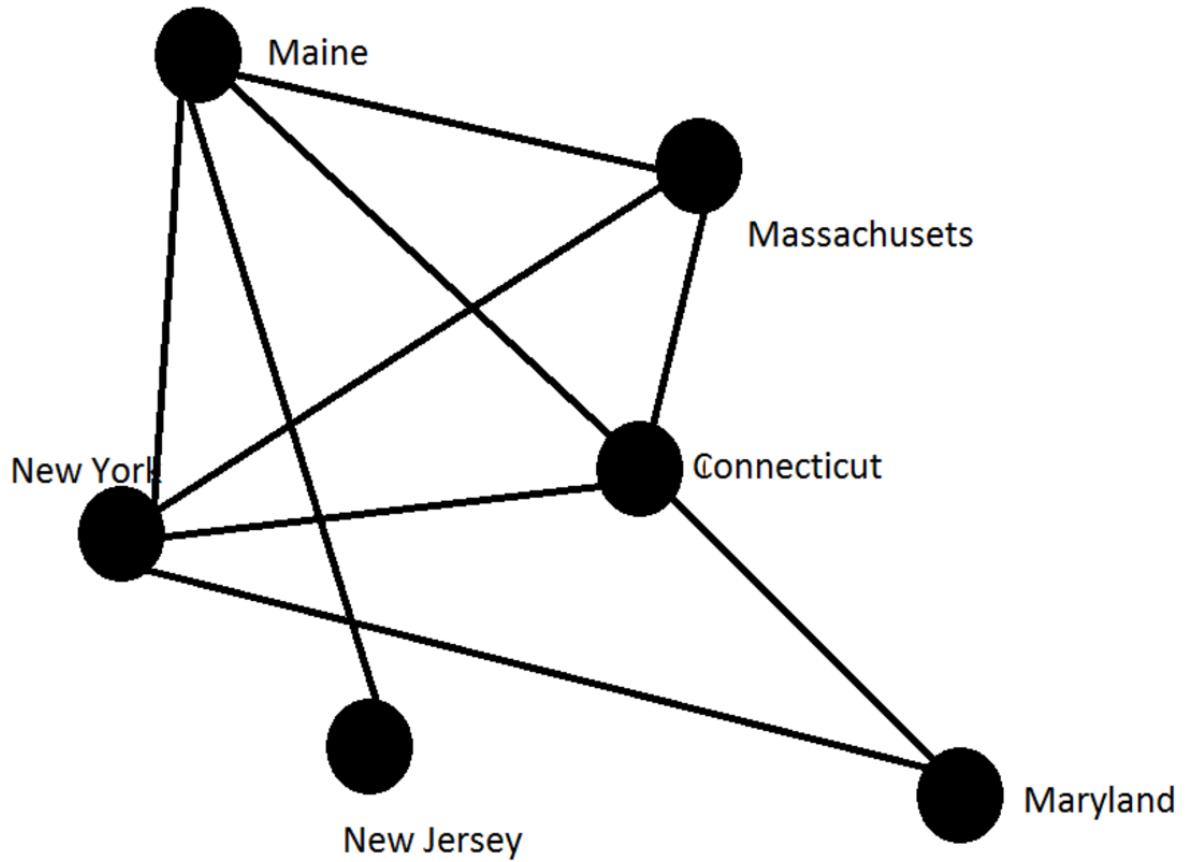
There are many systems in the natural world and in society that are amenable to mathematical and computational modeling. However, not everything is easily codified as a system of particles with coordinates and momenta. Some systems and problems such as social networks, ecologies, and genetic regulatory schemes are intrinsically divorced from spacetime descriptions, and instead are more naturally expressed as graphs that reflect their topological properties. At their simplest, graphs are simply collections of nodes – representing some class of objects like people, corporate boards, proteins, or destinations on the globe – and edges, which serve to represent connections like friendships, bridges, or molecular binding interactions.



## Contents

- What is a graph?
- Representation of Graphs
- Breadth-first Search
- Depth-first Search
- Contrasting Traversals
- Additional Problems

## What is a graph?



Consider the highway system of the eastern coast of the United States. A road inspector is given the task of writing reports about the current condition of each highway. What would be the most economical way for him to traverse all the cities? The problem can be modeled as a graph.

In fact, since graphs are dots and lines , they look like road maps. The dots are called vertices or nodes and the lines are called edges. They may have a value assigned to them (weighted) or they may just be a mere indicator of the existence of a path (unweighted). More formally, a graph can be defined as follows:

A graph  $G$  consists of a set of  $V$  of *vertices* (or *nodes*) and a set  $E$  of edges (or *arcs*) such that each edge  $e \in E$  is associated with a pair of vertices  $v \in V$ . A graph  $G$  with vertices  $V$  and edges  $E$  is written as  $G = (V, E)$ .

Because graphs are so pervasive, it is useful to define different types of graphs. The following are the most common types of graphs:

**Undirected graph:** An undirected graph is one in which its edges have no orientations, i.e. no direction is associated with any edge. The edges  $(x,y)$  and  $(y,x)$  are equivalent.

**Directed graph:** A directed graph or digraph  $GG$  consists of a set  $VV$  of vertices (or nodes) and a set of edges (or arcs) such that each edge  $e \in E$  is associated with an ordered pair of vertices. If there is an edge  $(x,y)$ , it is completely distinct from the edge  $(y,x)$ .



Undirected graphs are typically represented by a line with no arrows, which implies a bidirectional relationship between node A and node B. Directed graphs use an arrow to show the relationship from A to B.

<<<<< HEAD:wk-20-notes/general-notes-on-graphs/README.md

---

**Directed acyclic graph:** A directed acyclic graph (commonly abbreviated as DAG) is a directed graph with no directed cycles. A cycle is any path  $\{A_1, \dots, A_n\}$  such that the edges  $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_n \rightarrow A_1$  all exist, thus forming a loop. A DAG is a graph without a single cycle.

**Directed acyclic graph:** A directed acyclic graph (commonly abbreviated as DAG) is a directed graph with no directed cycles. A cycle is any path  $\{A_1, \dots, A_n\}$  such that the edges  $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_n \rightarrow A_1$  all exist, thus forming a loop. A DAG is a graph without a single cycle.

e4bf9b77d4b065ed20f39ffb8a1f8425c6ab66cf:python-language/general-notes-on-graphs/README.md

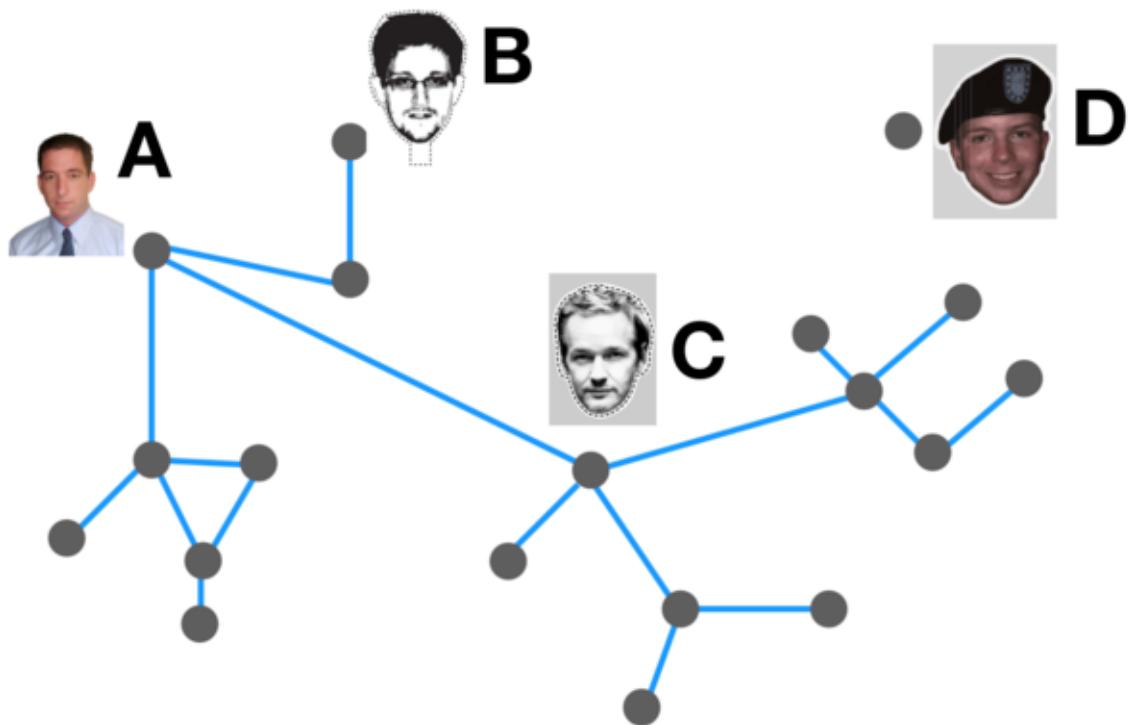
List all the edges and vertices of the undirected graph  $GG$  in the figure above.

The graph  $GG$  consists of the set of vertices  $VV = \{\text{Massachusetts, Maine, Connecticut, New York, Maryland, New Jersey}\}$ .

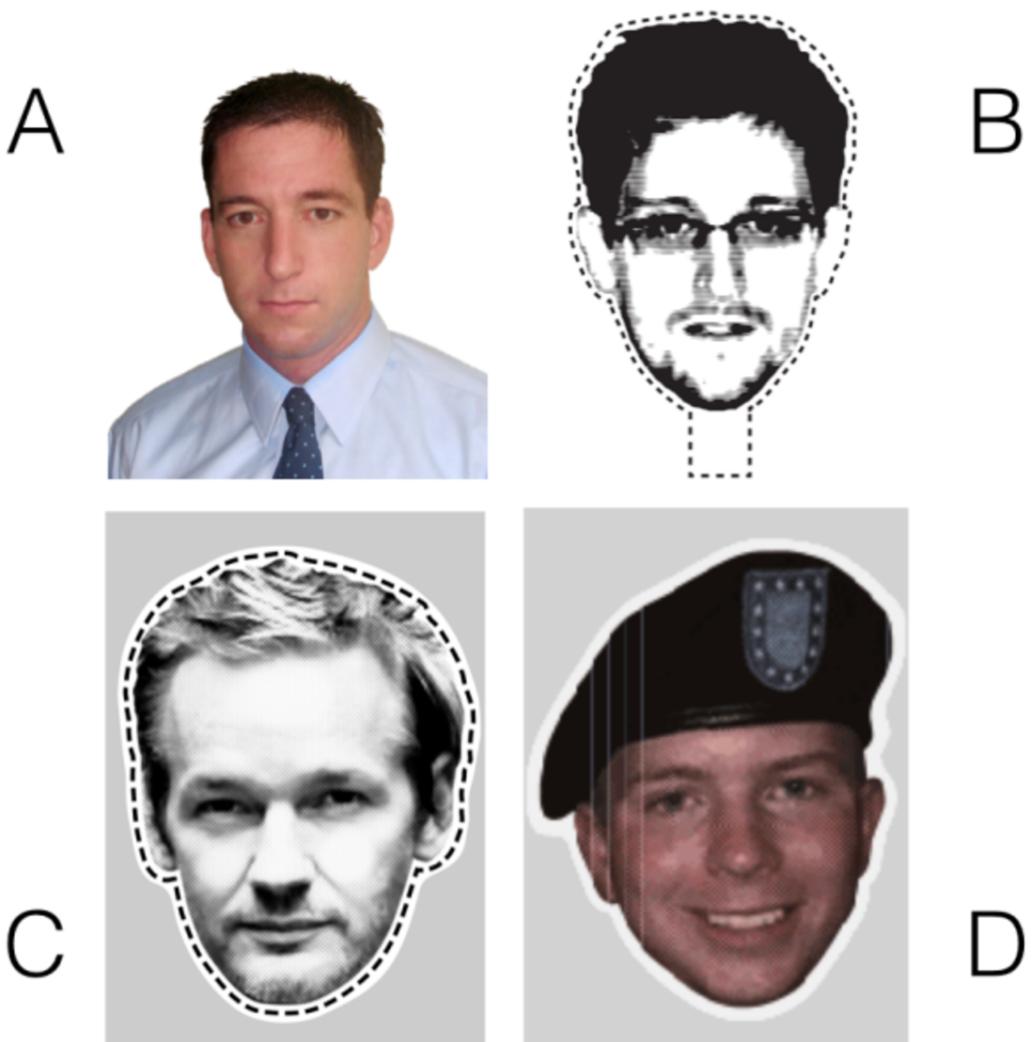
Its edges are  $E = \{(Maine, Massachusetts), (Massachusetts, Connecticut), (Connecticut, New York), (New York, Maine), (New York, Massachusetts), (New Jersey, Maine), (Maryland, New York), (Maine, Maryland)\}$ .

Note that since the graph is undirected, the order of the tuples in denoting the edges is unimportant.  $\blacksquare$

ADBC



Government surveillance agencies have a tendency to accumulate strange new powers during times of panic. The US National Security Agency (NSA) now has the ability to monitor the communications of suspected individuals as well as the communications of people within some number of hops of any suspect. In the communication network above, which person is connected to the greatest number of people through 1 hop or less?



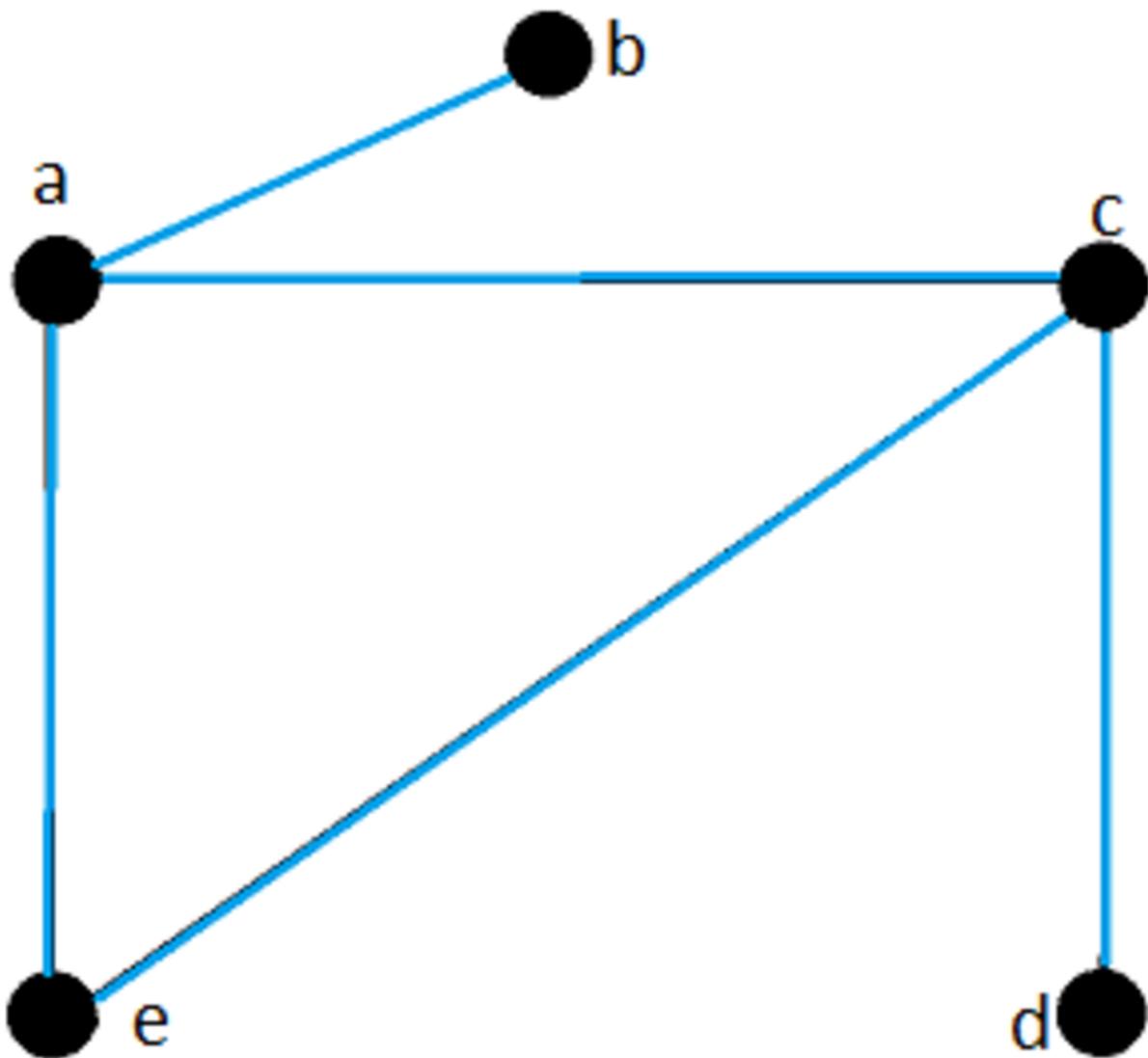
#### **Details and Assumptions:**

- Each dot represents a person.
- Each line represents communication between the people on either end.
- If X communicates with Y, and Y communicates with Z, we say that X and Z have a 1-hop connection, and that X has a 0-hop connection with Y.

#### **Representation of Graphs**

Above we represented a graph by drawing it. To represent it in a computer, however, we need more formal ways of representing graphs. Here we discuss the two most common ways of representing a graph: the adjacency matrix and the adjacency list.

#### **The adjacency matrix**



Represent the graph above using an adjacency matrix.

To obtain the adjacency matrix of the graph, we first label the rows and columns with the corresponding ordered vertices. If there exists an edge between two vertices  $i$  and  $j$ , then their corresponding cell in the matrix will be assigned 1. If there does not exist an edge, then the cell will be assigned the value 0. The adjacency matrix for the graph above is thus

\quad \begin{bmatrix} & a & b & c & d & e \\ a & 0 & 1 & 1 & 0 & 1 \\ b & 1 & 0 & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 1 & 0 & 1 \\ e & 1 & 0 & 0 & 1 & 0 \end{bmatrix}. \quad \square

### Adjacency list

An adjacency list representation of a graph is a way of associating each vertex (or node) in the graph with its respective list of neighboring vertices. A common way to do this is to create a

Hash table. This table will contain each vertex as a key and the list of adjacent vertices of that vertex as a value.

For our example above, the adjacency list representation will look as follows:

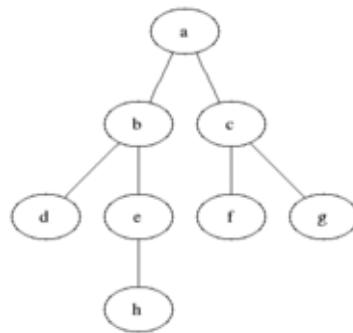
a	[b,e,c]
b	[a]
c	[a,d,e]
d	[c]
e	[c,a]

We can see that the adjacency list is much less expensive on memory as the adjacency matrix is very sparse.

Most graph algorithms involve visiting each vertex in VV, starting from a root node v\_0v0. There are several ways of achieving this. The two most common traversal algorithms are breadth-first search and depth-first search.

### Breadth-first Search

In a breadth-first search, we start with the start node, followed by its adjacent nodes, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on. Formally the BFS algorithm visits all vertices in a graph  $G$ , that are  $k$  edges away from the source vertex  $s$  before visiting any vertex  $k+1$  edges away. This is done until no more vertices are reachable from  $s$ . The image below demonstrates exactly how this traversal proceeds:



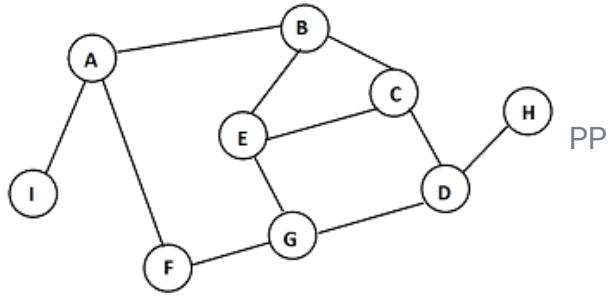
For a graph  $G = (V, E)$  and a source vertex  $v$ , breadth-first search traverses the edges of  $G$  to find all reachable vertices from  $v$ . It also computes the shortest distance to any reachable vertex. Any path between two points in a breadth-first search tree corresponds to the shortest path from the root  $v$  to any other node  $s$ .

---

We may think of three types of vertices in BFS as *tree* vertices, those that have been taken off the data structure. *fringe* vertices, those adjacent to tree vertices but not yet visited, and *undiscovered* vertices, those that we have not encountered yet. If each visited vertex is connected to the edge that caused it to be added to the data structure, then these edges form a tree.

To search a connected component of a graph systematically, we begin with one vertex on the fringe, all others unseen, and perform the following step until all vertices have been visited: "move one vertex  $x$  from the fringe to the tree, and put any unseen vertices adjacent to  $x$  on the fringe." Graph traversal methods differ in how it is decided which vertex should be moved from the fringe to the tree. For breadth-first search we want to choose the vertex from the fringe that was least recently encountered; this corresponds to using a queue to hold vertices on the fringe.

What is the state of the queue at each iteration of BFS, if it is called from node 'a'?



The table below shows the contents of the queue as the procedure. BFS visits vertices in the graph above. BFS will visit the same vertices as DFS. In this example all of them.

<<<<< HEAD:wk-20-notes/general-notes-on-graphs/README.md

```

\begin{array}{l|l}
\textbf{Node Visited} & \textbf{Queue} \\
\hline
a & a \\
\text{(empty)} & b \\
b & f \\
\text{(empty)} & b f \\
i & i \\
b f i & \\
\text{(empty)} & f i \\
c & c \\
f i c & e \\
f i c e & g \\
i c e g & \\
e g & d \\
e g d & \\
g d & \\
d & \\
\text{(empty)} & h \\
h & \\
\text{(empty)} & \\
\end{array}

```

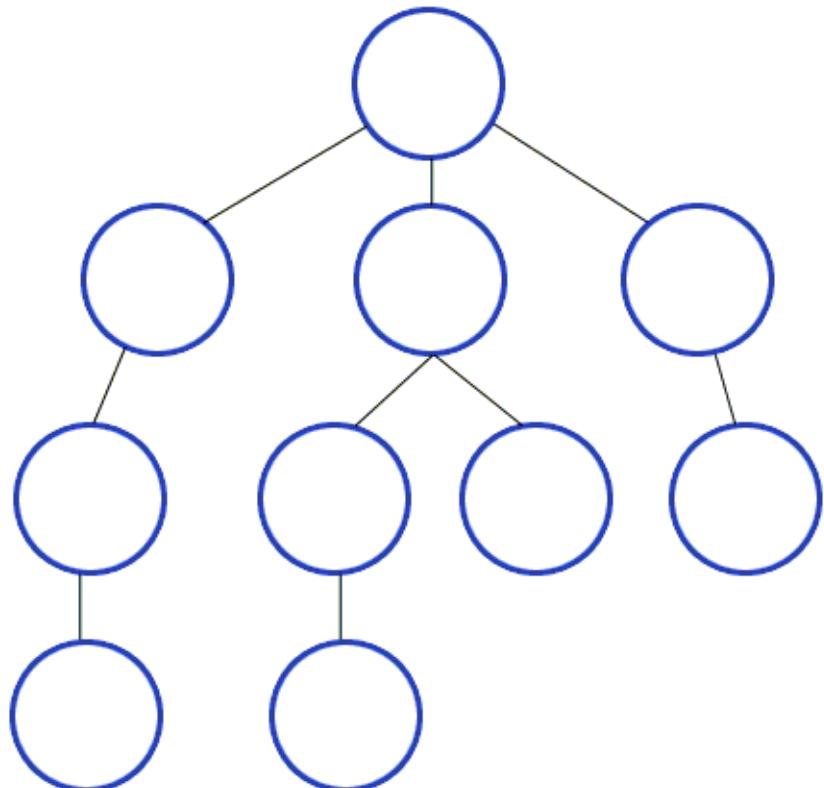
Visited nodes: a b f i c e g d h  
 Queue: a (empty) b b f if if i c f i c e i c e i c e g c e g e g e g d g d d  
 (empty) h (empty)



## Depth-first Search

Depth-first search explores edges out of the most recently discovered vertex  $s$  that still has unexplored edges leaving it. Once all of  $s$ 's edges have been explored, the search “backtracks” to explore edges leaving the vertex from which  $s$  was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex:

- Visit a vertex  $s$ .
- Mark  $s$  as visited.
- Recursively visit each unvisited vertex attached to  $s$ .



A recursive implementation of DFS:

---

A non-recursive implementation of DFS, it delays whether a vertex has been discovered until the vertex has been popped from the stack.

---

## Contrasting Traversals

Similar to tree traversal, the code for breadth-first search is slightly different from depth-first search. The most commonly mentioned difference is that BFS uses a queue to store alternative choices instead of a stack. This small change however leads to a classical graph traversal algorithm. Depth-first search goes down a single path until the path leads to the goal or we reach a limit. When a path is completely explored we back track. BFS however explores all paths from the starting location at the same time.

As we increase the size of our graph, the contrast between depth-first and breadth-first search is quite evident. Depth-first search explores the graph by looking for new vertices far away from the start point, taking closer vertices only when dead ends are encountered; breadth-first search completely covers the area close to the starting point, moving farther away only when everything close has been looked at. Again, the order in which the nodes are visited depends largely upon the effects of this ordering on the order in which vertices appear on the adjacency lists.

## Additional Problems



John lives in the Trees of Ten Houses, and it is a most

ideal and idyllic place for him and the other dwellers up in the canopy. They have invested a tremendous amount of time in engineering these houses, and to ensure no house felt isolated from the others, they built a fresh, finely crafted bridge between each and every house!

Unfortunately, the Trees of Ten Houses were not immune to thunderstorms, nor were the bridges well engineered. The night was treacherous, howling with wind and freezing with rain, so the odds for the bridges were not good—each bridge seemed just as likely to survive as to be shattered!

Fortunately, as there were so very many bridges in the Trees of Ten Houses, when John did wake the following morning, he found he was able to make his way to each and every house using only the existing bridges, though round-about routes may have been necessary. As they began rebuilding, John became curious... what were the chances that they'd all be so lucky?

More formally, if  $P$  is the probability that, after the storm, John is able to traverse to each and every house, what is  $\lfloor 10^{10} P \rfloor$ ?

#### **Details and Assumptions:**

- The Trees of Ten Houses do, in fact, contain precisely 10 houses.
- Before the storm, there exists a single bridge between each and every unique pair of houses.
- The storm destroys each bridge with independent probability  $\frac{1}{2}$ .
- John is allowed to traverse through others' houses to try to reach all of them, but he must only use the surviving bridges to get there. No vine swinging allowed.

\*\*\*\*

# Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below *tut-private*), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

---

## A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This

is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

---

## Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion —users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace!

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, *forgetting* would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared `global`, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global or nonlocal statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

## Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
1 def scope_test():
2 def do_local():
3 spam = "local spam"
4
5 def do_nonlocal():
6 nonlocal spam
7 spam = "nonlocal spam"
8
9 def do_global():
10 global spam
11 spam = "global spam"
12
13 spam = "test spam"
14 do_local()
15 print("After local assignment:", spam)
16 do_nonlocal()
17 print("After nonlocal assignment:", spam)
18 do_global()
19 print("After global assignment:", spam)
```

```
20
21 scope_test()
22 print("In global scope:", spam)
```

The output of the example code is:

```
``` {.sourceCode .none} After local assignment: test spam After nonlocal assignment: nonlocal
spam After global assignment: nonlocal spam In global scope: global spam
```

```

Note how the *local* assignment (which is default) didn't change *scope\_test*'s binding of *spam*. The nonlocal assignment changed *scope\_test*'s binding of *spam*, and the global assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the global assignment.

---

## A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### Class Definition Syntax

The simplest form of class definition looks like this:

```
1 class ClassName:
2 <statement-1>
3 .
4 .
5 .
6 <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

## Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python:

`obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
1 class MyClass:
2 """A simple example class"""
3 i = 12345
4
5 def f(self):
6 return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__`, like this:

```
1 def __init__(self):
2 self.data = []
```

When a class defines an `__init__` method, class instantiation automatically invokes `__init__` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__`. For example, :

```
class Complex: ... def __init__(self, realpart, imagpart): ... self.r = realpart ... self.i = imagpart
... x = Complex(3.0, -4.5) x.r, x.i (3.0, -4.5)
```

## Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
1 x.counter = 1
2 while x.counter < 10:
3 x.counter = x.counter * 2
4 print(x.counter)
5 del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

## Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
1 xf = x.f
2 while True:
3 print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of  $n$  arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

## Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
1 class Dog:
2
3 kind = 'canine' # class variable shared by all instances
4
5 def __init__(self, name):
6 self.name = name # instance variable unique to each instance
7
8 >>> d = Dog('Fido')
9 >>> e = Dog('Buddy')
10 >>> d.kind # shared by all dogs
11 'canine'
12 >>> e.kind # shared by all dogs
13 'canine'
14 >>> d.name # unique to d
15 'Fido'
16 >>> e.name # unique to e
17 'Buddy'
```

As discussed in tut-object, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
1 class Dog:
2
3 tricks = [] # mistaken use of a class variable
4
5 def __init__(self, name):
6 self.name = name
7
8 def add_trick(self, trick):
9 self.tricks.append(trick)
10
11 >>> d = Dog('Fido')
12 >>> e = Dog('Buddy')
13 >>> d.add_trick('roll over')
14 >>> e.add_trick('play dead')
15 >>> d.tricks # unexpectedly shared by all dogs
16 ['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
1 class Dog:
2
3 def __init__(self, name):
4 self.name = name
5 self.tricks = [] # creates a new empty list for each dog
6
7 def add_trick(self, trick):
8 self.tricks.append(trick)
9
10 >>> d = Dog('Fido')
11 >>> e = Dog('Buddy')
12 >>> d.add_trick('roll over')
13 >>> e.add_trick('play dead')
14 >>> d.tricks
15 ['roll over']
16 >>> e.tricks
17 ['play dead']
```

## Random Remarks

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
class Warehouse: purpose = 'storage' region = 'west'

w1 = Warehouse() print(w1.purpose, w1.region) storage west
w2 = Warehouse()
w2.region = 'east' print(w2.purpose, w2.region) storage east
```

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided —again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
1 # Function defined outside the class
2 def f1(self, x, y):
```

```
3 return min(x, x+y)
4
5 class C:
6 f = f1
7
8 def g(self):
9 return 'hello world'
10
11 h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
1 class Bag:
2 def __init__(self):
3 self.data = []
4
5 def add(self, x):
6 self.data.append(x)
7
8 def addtwice(self, x):
9 self.add(x)
10 self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

# Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
1 class DerivedClassName(BaseClassName):
2 <statement-1>
3 .
4 .
5 .
6 <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method

directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

## Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
1 class DerivedClassName(Base1, Base2, Base3):
2 <statement-1>
3 .
4 .
5 .
6 <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to super. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see <https://www.python.org/download/releases/2.3/mro/>.

---

## Private Variables

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
1 class Mapping:
2 def __init__(self, iterable):
3 self.items_list = []
4 self.__update(iterable)
5
6 def update(self, iterable):
7 for item in iterable:
```

```

8 self.items_list.append(item)
9
10 __update = update # private copy of original update() method
11
12 class MappingSubclass(Mapping):
13
14 def update(self, keys, values):
15 # provides new signature for update()
16 # but does not break __init__()
17 for item in zip(keys, values):
18 self.items_list.append(item)

```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

## Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a few named data items. An empty class definition will do nicely:

```

1 class Employee:
2 pass
3
4 john = Employee() # Create an empty employee record
5
6 # Fill the fields of the record
7 john.name = 'John Doe'
8 john.dept = 'computer lab'

```

```
9 john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read` and `!readline` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m`, and `m.__func__` is the function object corresponding to the method.

## Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
1 for element in [1, 2, 3]:
2 print(element)
3 for element in (1, 2, 3):
4 print(element)
5 for key in {'one':1, 'two':2}:
6 print(key)
7 for char in "123":
8 print(char)
9 for line in open("myfile.txt"):
10 print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter` on the container object. The function returns an iterator object that defines the method `~iterator.__next__` which accesses elements in the container one at a time. When there are no more elements, `~iterator.__next__` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `~iterator.__next__` method using the `next` built-in function; this example shows how it all works:

```
s = 'abc'
it = iter(s)
it next(it) 'a'
next(it) 'b'
next(it) 'c'
next(it)
Traceback (most recent call
last): File "", line 1, in next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__` method which returns an object with a `__next__` method. If the class defines `__next__`, then `__iter__` can just return `self`:

```
1 class Reverse:
2 """Iterator for looping over a sequence backwards."""
3 def __init__(self, data):
4 self.data = data
5 self.index = len(data)
6
7 def __iter__(self):
8 return self
9
10 def __next__(self):
11 if self.index == 0:
12 raise StopIteration
13 self.index = self.index - 1
14 return self.data[self.index]
15
16 >>> rev = Reverse('spam')
17 >>> iter(rev)
18 <__main__.Reverse object at 0x00A1DB50>
19 >>> for char in rev:
20 ... print(char)
21 ...
22 m
23 a
24 p
25 s
```

## Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
1 def reverse(data):
2 for index in range(len(data)-1, -1, -1):
3 yield data[index]
4
```

```
5 >>> for char in reverse('golf'):
6 ...
7 print(char)
8
9 f
10 l
11 o
12 g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__` and `__next__` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

---

## Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
sum(i*i for i in range(10)) # sum of squares 285
```

```
xvec = [10, 20, 30] yvec = [7, 5, 3] sum(x*y for x,y in zip(xvec, yvec)) # dot product 260
```

```
unique_words = set(word for line in page for word in line.split())
```

```
valedictorian = max((student.gpa, student.name) for student in graduates)
```

```
| | | data = 'golf' list(data[i] for i in range(len(data)-1, -1, -1)) ['f', 'l', 'o', 'g']
```

## Footnotes

1 attribute called `\~object.\_\_dict\_\_` which returns the dictionary  
2 used to implement the module's namespace; the name  
3 `\~object.\_\_dict\_\_` is an attribute but not a global name.  
4 Obviously, using this violates the abstraction of namespace  
5 implementation, and should be restricted to things like post-mortem  
6 debuggers.

# Collections In Python

## What Are Collections In Python?

Collections in python are basically container data types, namely lists, sets, tuples, dictionary. They have different characteristics based on the declaration and usage.

- A list is declared in square brackets, it is mutable, stores duplicate values and elements can be accessed using indexes.
- A tuple is ordered and immutable in nature, although duplicate entries can be there inside a tuple.
- A set is unordered and declared in square brackets. It is not indexed and does not have duplicate entries as well.
- A dictionary has key-value pairs and is mutable in nature. We use square brackets to declare a dictionary.

These are the python's general-purpose built-in container data types. But as we all know, python always has a little something extra to offer. It comes with a python module named collections which has specialized data structures.

---

## Specialized Collection Data Structures

Collections module in python implements specialized data structures which provide an alternative to python's built-in container data types. Following are the specialized data structures in the collections module.

1. namedtuple()
2. deque
3. Chainmap
4. Counter
5. OrderedDict
6. defaultdict
7. UserDict
8. UserList
9. UserString

## **namedtuple( )**

It returns a tuple with a named entry, which means there will be a name assigned to each value in the tuple. It overcomes the problem of accessing the elements using the index values. With namedtuple( ) it becomes easier to access these values since you do not have to remember the index values to get specific elements.

### **How It Works?**

First of all, you must import the collections module, it does not require installation.

```
from collections import namedtuple
```

Look at the following code to understand how you can use namedtuple.

```
1 a = namedtuple('courses' , 'name , tech')
2 s = a('data science' , 'python')
3 print(s)
4
5 #the output will be courses(name='python' , tech='python')
```

### **How To Create A namedtuple Using A List?**

```
1 s._make(['data science' , 'python'])
2 #the output will be same as before.
```

## **deque**

deque pronounced as 'deck' is an optimized list to perform insertion and deletion easily.

### **How it works?**

```
1 #creating a deque
```

```
2 from collections import deque
3
4 a = ['d' , 'u' , 'r' , 'e' , 'k']
5 a1 = deque(a)
6 print(a1)
7 #the output will be deque(['d' , 'u' , 'r' , 'e' , 'k'])
```

Now let's take a look at how we will insert and remove items from deque.

```
1 a1.append('a')
2 print(a1)
3 # the output will be deque(['d' , 'u' , 'r' , 'e' , 'k' , 'a'])
4 a1.appendleft('e')
5 print(a1)
6 # the output will be deque(['e' , 'd' , 'u' , 'r' , 'e' , 'k' , 'a'])
```

As should be obvious, inserting a component is enhanced utilizing deque, also you can remove components as well.

```
1 a1.pop()
2 print(a1)
3 #the output will be deque(['e' , 'd' , 'u' , 'r' , 'e' , 'k'])
4 a1.popleft()
5 print(a1)
6 #the output will be deque(['d' , 'u' , 'r' , 'e' , 'k'])
```

Similar to the built-in data types, there are several other operations that we can perform on a deque. Like counting elements or clearing the deque etc.

## ChainMap

It is a dictionary-like class which is able to make a single view of multiple mappings. It basically returns a list of several other dictionaries. Suppose you have two dictionaries with several key-value pairs, in this case, ChainMap will make a single list with both the dictionaries in it.

### How it works?

```
1 from collections import ChainMap
2 a = { 1: 'edureka' , 2: 'python'}
3 b = {3: 'data science' , 4: 'Machine learning'}
4 c = ChainMap(a,b)
5 print(c)
6 #the output will be ChainMap[{1: 'edureka' , 2: 'python'} , {3: 'data science' , 4: 'Machine learning'}]
```

To access or insert elements we use the keys as an index. But to add a new dictionary in the ChainMap we use the following approach.

```
1 a1 = { 5: 'AI' , 6: 'neural networks'}
2 c1 = c.new_child(a1)
3 print(c1)
4 #the output will be ChainMap[{1: 'edureka' , 2: 'python'} , {3: 'data science' , 4: 'Machine learning'} , {5: 'AI' , 6: 'neural networks'}]
```

## Counter

It is a dictionary subclass which is used to count hashable objects.

### How it works?

```
1 from collections import Counter
2 a = [1,1,1,1,2,3,3,4,3,3,4]
3 c = Counter(a)
4 print(c)
5 #the output will be Counter = ({1:4 , 2:1 , 3:4 , 4:2})
```

In addition to the operations, you can perform on a dictionary Counter has 3 more operations that we can perform.

1. element function – It returns a list containing all the elements in the Counter.
2. Most\_common() – It returns a sorted list with the count of each element in the Counter.
3. Subtract( ) – It takes an iterable object as an argument and deducts the count of the elements in the Counter.

## OrderedDict

It is a dictionary subclass which remembers the order in which the entries were added. Basically, even if you change the value of the key, the position will not be changed because of the order in which it was inserted in the dictionary.

### How it works?

```
1 from collections import OrderedDict
2 od = OrderedDict()
3 od[1] = 'e'
4 od[2] = 'd'
5 od[3] = 'u'
6 od[4] = 'r'
7 od[5] = 'e'
8 od[6] = 'k'
9 od[7] = 'a'
10 print(od)
11 #the output will be OrderedDict[(1 , 'e') , (2 , 'd') , (3 , 'u') , (4 , 'r')],
```

It does not matter what value gets inserted in the dictionary, the OrderedDict remembers the order in which it was inserted and gets the output accordingly. Even if we change the value of the key. Let's say, if we change the key-value 4 to 8, the order will not change in the output.

## defaultdict

It is a dictionary subclass which calls a factory function to supply missing values. In general, it does not throw any errors when a missing key value is called in a dictionary.

### How it works?

```
1 from collections import defaultdict
2 d = defaultdict(int)
3 #we have to specify a type as well.
4 d[1] = 'edureka'
5 d[2] = 'python'
6 print(d[3])
7 #it will give the output as 0 instead of keyerror.
```

## UserDict

This class acts as a wrapper around dictionary objects. The need for this class came from the necessity to subclass directly from dict. It becomes easier to work with this class as the underlying dictionary becomes an attribute.

```
class collections.UserDict([initialdata])
```

This class simulates a dictionary. The content of the instance are kept in a regular dictionary which can be accessed with the 'data' attribute of the class UserDict. The reference of initial data is not kept, for it to be used for other purposes.

## UserList

This class acts like a wrapper around the list-objects. It is a useful base class for other lists like classes which can inherit from them and override the existing methods or even add fewer new ones as well.

The need for this class came from the necessity to subclass directly from the list. It becomes easier to work with this class as the underlying list becomes an attribute.

```
class collections.UserList([list])
```

It is the class that simulates a list. The contents of the instance are kept in a customary list. The sub-classes of the list are relied upon to offer a constructor which can be called with either no or one contention.

reference

# Untitled

DATA\_STRUC\_PYTHON\_NOTES/BST.ipynb at

97785caf08ed224f4eff0b08fdd8d084f276d819 · bgoonz/DATA\_STRUC\_PYTHON\_NOTES

[https://github.com/bgoonz/DATA\\_STRUC\\_PYTHON\\_NOTES/blob/97785caf08ed224f4eff0b08fdd8d084f276d819/\\_WEEKS/wk19/d3/BST.ipynb](https://github.com/bgoonz/DATA_STRUC_PYTHON_NOTES/blob/97785caf08ed224f4eff0b08fdd8d084f276d819/_WEEKS/wk19/d3/BST.ipynb)

**guides-n-tutorials**

# Problems w Solutions

- Python built-in Modules [ 31 Exercises with Solution ]
- Python Data Types - String [ 101 Exercises with Solution ]
- Python JSON [ 9 Exercises with Solution ]
- Python Data Types - List [ 272 Exercises with Solution ]
- Python Data Types - Dictionary [ 80 Exercises with Solution ]
- Python Data Types - Tuple [ 33 Exercises with Solution ]
- Python Data Types - Sets [ 20 Exercises with Solution ]
- Python Data Types - Collections [ 36 Exercises with Solution ]
- Python heap queue algorithm [ 29 exercises with solution ]
- Python Array [ 24 Exercises with Solution ]
- Python Enum [ 5 Exercises with Solution ]
- Python Bisect [ 9 Exercises with Solution ]
- Python Conditional statements and loops [ 44 Exercises with Solution]
- Python functions [ 21 Exercises with Solution ]
- Python Lambda [ 52 Exercises with Solution ]
- Python Map [ 17 Exercises with Solution ]
- Python Operating System Services [ 18 Exercises with Solution ]
- Python Date Time [ 63 Exercises with Solution ]
- Python Class [ 24 Exercises with Solution ]
- Search and Sorting [ 39 Exercises with Solution ]
- Linked List [ 14 Exercises with Solution ]
- Binary Search Tree [ 6 Exercises with Solution ]
- Recursion [ 11 Exercises with Solution ]
- Python Math [ 88 Exercises with Solution ]
- Python File Input Output [ 21 Exercises with Solution ]
- Python Regular Expression [ 56 Exercises with Solution ]
- Python SQLite Database [ 13 Exercises with Solution ]
- Python CSV File Reading and Writing [ 11 exercises with solution ]
- Python Itertools [ 44 exercises with solution ]
- Python Requests [ 9 exercises with solution ]
- More to Come !

## Python GUI tkinter

- Python tkinter Basic [ 5 Exercises with Solution ]
- Python tkinter widgets [ 12 Exercises with Solution ]

## **Python NumPy :**

- Python NumPy Home
- Python NumPy Basic [ 59 Exercises with Solution ]
- Python NumPy arrays [ 205 Exercises with Solution ]
- Python NumPy Linear Algebra [ 19 Exercises with Solution ]
- Python NumPy Random [ 17 Exercises with Solution ]
- Python NumPy Sorting and Searching [ 9 Exercises with Solution ]
- Python NumPy Mathematics [ 41 Exercises with Solution ]
- Python NumPy Statistics [ 14 Exercises with Solution ]
- Python NumPy DateTime [ 7 Exercises with Solution ]
- Python NumPy String [ 22 Exercises with Solution ]
- More to come

## **Python Challenges :**

- Python Challenges: Part -1 [ 1- 64 ]
- More to come

## **Python Mini Projects :**

- Python Projects Numbers: [ 11 Projects with solution ]
- Python Web Programming: [ 12 Projects with solution ]
- Python Projects: Novel Coronavirus (COVID-19) [ 14 Exercises with Solution ]
- More to come

## **Python Pandas :**

- Python Pandas Home
- Pandas Data Series [ 40 exercises with solution ]
- Pandas DataFrame [ 81 exercises with solution ]
- Pandas Index [ 26 exercises with solution ]
- Pandas String and Regular Expression [ 41 exercises with solution ]
- Pandas Joining and merging DataFrame [ 15 exercises with solution ]
- Pandas Grouping and Aggregating [ 32 exercises with solution ]
- Pandas Time Series [ 32 exercises with solution ]
- Pandas Filter [ 27 exercises with solution ]
- Pandas GroupBy [ 32 exercises with solution ]
- Pandas Handling Missing Values [ 20 exercises with solution ]

- Pandas Style [ 15 exercises with solution ]
- Pandas Excel Data Analysis [ 25 exercises with solution ]
- Pandas Pivot Table [ 32 exercises with solution ]
- Pandas Datetime [ 25 exercises with solution ]
- Pandas Plotting [ 19 exercises with solution ]
- Pandas SQL database Queries [ 24 exercises with solution ]
- Pandas IMDb Movies Queries [ 17 exercises with solution ]
- Pandas Practice Set-1 [ 65 exercises with solution ]

### **Python Machine Learning :**

- Python Machine learning Iris flower data set [38 exercises with solution]
- More to come

### **Learn Python packages using Exercises, Practice, Solution and explanation**

### **Python GeoPy Package :**

- Python GeoPy Package [ 7 exercises with solution ]

### **Python BeautifulSoup :**

- Python BeautifulSoup [ 36 exercises with solution ]

### **Python Arrow Module :**

- Python Arrow Module [ 27 exercises with solution ]

### **Python Web Scraping :**

- Python Web Scraping [ 27 Exercises with solution ]

### **List of Python Exercises :**

- Python Basic (Part -I) [ 150 Exercises with Solution ]
- Python Basic (Part -II) [ 142 Exercises with Solution ]

*An editor is available at the bottom of the page to write and execute the scripts.]*

**1.** Write a Python function that takes a sequence of numbers and determines whether all the numbers are different from each other. Go to the editor

[Click me to see the sample solution](#)

**2.** Write a Python program to create all possible strings by using 'a', 'e', 'i', 'o', 'u'. Use the characters exactly once. Go to the editor

[Click me to see the sample solution](#)

**3.** Write a Python program to remove and print every third number from a list of numbers until the list becomes empty.

[Click me to see the sample solution](#)

**4.** Write a Python program to find unique triplets whose three elements gives the sum of zero from an array of n integers. Go to the editor

[Click me to see the sample solution](#)

**5.** Write a Python program to create the combinations of 3 digit combo. Go to the editor

[Click me to see the sample solution](#)

**6.** Write a Python program to print a long text, convert the string to a list and print all the words and their frequencies. Go to the editor

[Click me to see the sample solution](#)

**7.** Write a Python program to count the number of each character of a given text of a text file.

[Go to the editor](#)

[Click me to see the sample solution](#)

**8.** Write a Python program to get the top stories from Google news. Go to the editor

[Click me to see the sample solution](#)

**9.** Write a Python program to get a list of locally installed Python modules. Go to the editor

[Click me to see the sample solution](#)

**10.** Write a Python program to display some information about the OS where the script is running. Go to the editor

[Click me to see the sample solution](#)

**11.** Write a Python program to check the sum of three elements (each from an array) from three arrays is equal to a target value. Print all those three-element combinations. Go to the editor

Sample data:

```
/*
X = [10, 20, 20, 20]
Y = [10, 20, 30, 40]
Z = [10, 30, 40, 20]
target = 70
*/
```

[Click me to see the sample solution](#)

**12.** Write a Python program to create all possible permutations from a given collection of distinct numbers.[Go to the editor](#)

[Click me to see the sample solution](#)

**13.** Write a Python program to get all possible two digit letter combinations from a digit (1 to 9) string. [Go to the editor](#)

```
string_maps = {
 "1": "abc",
 "2": "def",
 "3": "ghi",
 "4": "jkl",
 "5": "mno",
 "6": "pqrs",
 "7": "tuv",
 "8": "wxy",
 "9": "z"
}
```

[Click me to see the sample solution](#)

**14.** Write a Python program to add two positive integers without using the '+' operator. [Go to the editor](#)

Note: Use bit wise operations to add two numbers.

[Click me to see the sample solution](#)

**15.** Write a Python program to check the priority of the four operators (+, -, \*, /). [Go to the editor](#)

[Click me to see the sample solution](#)

**16.** Write a Python program to get the third side of right angled triangle from two given sides.

[Go to the editor](#)

[Click me to see the sample solution](#)

**17.** Write a Python program to get all strobogrammatic numbers that are of length n. Go to the editor

A strobogrammatic number is a number whose numeral is rotationally symmetric, so that it appears the same when rotated 180 degrees. In other words, the numeral looks the same right-side up and upside down (e.g., 69, 96, 1001).

For example,

Given n = 2, return ["11", "69", "88", "96"].

Given n = 3, return ['818', '111', '916', '619', '808', '101', '906', '609', '888', '181', '986', '689']  
Click me to see the sample solution

**18.** Write a Python program to find the median among three given numbers. Go to the editor

Click me to see the sample solution

**19.** Write a Python program to find the value of n where n degrees of number 2 are written sequentially in a line without spaces. Go to the editor

Click me to see the sample solution

**20.** Write a Python program to find the number of zeros at the end of a factorial of a given positive number. Go to the editor

Range of the number(n): (1 <= n <= 2\*109).

Click me to see the sample solution

**21.** Write a Python program to find the number of notes (Sample of notes: 10, 20, 50, 100, 200 and 500 ) against a given amount. Go to the editor

Range - Number of notes(n) : n (1 <= n <= 1000000).

Click me to see the sample solution

**22.** Write a Python program to create a sequence where the first four members of the sequence are equal to one, and each successive term of the sequence is equal to the sum of the four previous ones. Find the Nth member of the sequence. Go to the editor

Click me to see the sample solution

**23.** Write a Python program that accept a positive number and subtract from this number the sum of its digits and so on. Continues this operation until the number is positive. Go to the editor

Click me to see the sample solution

**24.** Write a Python program to find the number of divisors of a given integer is even or odd. Go to the editor

[Click me to see the sample solution](#)

**25.** Write a Python program to find the digits which are absent in a given mobile number. Go to the editor

[Click me to see the sample solution](#)

**26.** Write a Python program to compute the summation of the absolute difference of all distinct pairs in a given array (non-decreasing order). Go to the editor

Sample array: [1, 2, 3]

Then all the distinct pairs will be:

1 2

1 3

2 3

[Click me to see the sample solution](#)

**27.** Write a Python program to find the type of the progression (arithmetic progression/geometric progression) and the next successive member of a given three successive members of a sequence. Go to the editor

According to Wikipedia, an arithmetic progression (AP) is a sequence of numbers such that the difference of any two successive members of the sequence is a constant. For instance, the sequence 3, 5, 7, 9, 11, 13, . . . is an arithmetic progression with common difference 2. For this problem, we will limit ourselves to arithmetic progression whose common difference is a non-zero integer.

On the other hand, a geometric progression (GP) is a sequence of numbers where each term after the first is found by multiplying the previous one by a fixed non-zero number called the common ratio. For example, the sequence 2, 6, 18, 54, . . . is a geometric progression with common ratio 3. For this problem, we will limit ourselves to geometric progression whose common ratio is a non-zero integer.

[Click me to see the sample solution](#)

**28.** Write a Python program to print the length of the series and the series from the given 3rd term, 3rd last term and the sum of a series. Go to the editor

Sample Data:

Input third term of the series: 3

Input 3rd last term: 3

Sum of the series: 15

Length of the series: 5

Series:

1 2 3 4 5

[Click me to see the sample solution](#)

**29.** Write a Python program to find common divisors between two numbers in a given pair. Go to the editor

[Click me to see the sample solution](#)

**30.** Write a Python program to reverse the digits of a given number and add it to the original, If the sum is not a palindrome repeat this procedure. Go to the editor

Note: A palindrome is a word, number, or other sequence of characters which reads the same backward as forward, such as madam or racecar.

[Click me to see the sample solution](#)

**31.** Write a Python program to count the number of carry operations for each of a set of addition problems. Go to the editor

According to Wikipedia " In elementary arithmetic, a carry is a digit that is transferred from one column of digits to another column of more significant digits. It is part of the standard algorithm to add numbers together by starting with the rightmost digits and working to the left. For example, when 6 and 7 are added to make 13, the "3" is written to the same column and the "1" is carried to the left".

[Click me to see the sample solution](#)

**32.** Write a python program to find heights of the top three building in descending order from eight given buildings. Go to the editor

**Input:**

0 <= height of building (integer) <= 10,000

Input the heights of eight buildings:

25

35

15

16

30

45

37

39

Heights of the top three buildings:

45

39

37

[Click me to see the sample solution](#)

**33.** Write a Python program to compute the digit number of sum of two given integers. Go to the editor

**Input:**

Each test case consists of two non-negative integers  $x$  and  $y$  which are separated by a space in a line.

$0 \leq x, y \leq 1,000,000$

Input two integers(a b):

5 7

Sum of two integers a and b.:

2

[Click me to see the sample solution](#)

**34.** Write a Python program to check whether three given lengths (integers) of three sides form a right triangle. Print "Yes" if the given sides form a right triangle otherwise print "No". Go to the editor

**Input:**

Integers separated by a single space.

$1 \leq \text{length of the side} \leq 1,000$

Input three integers(sides of a triangle)

8 6 7

No

[Click me to see the sample solution](#)

**35.** Write a Python program which solve the equation: Go to the editor

$ax+by=c$

$dx+ey=f$

Print the values of  $x$ ,  $y$  where  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$  and  $f$  are given.

**Input:**

$a,b,c,d,e,f$  separated by a single space.

$(-1,000 \leq a,b,c,d,e,f \leq 1,000)$

Input the value of  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$ ,  $f$ :

5 8 6 7 9 4

Values of  $x$  and  $y$ :

-2.000 2.000

[Click me to see the sample solution](#)

**36.** Write a Python program to compute the amount of the debt in n months. The borrowing amount is \$100,000 and the loan adds 5% interest of the debt and rounds it to the nearest 1,000 above month by month. Go to the editor

**Input:**

An integer n (0 <= n <= 100)

Input number of months: 7

Amount of debt: \$144000

[Click me to see the sample solution](#)

**37.** Write a Python program which reads an integer n and find the number of combinations of a,b,c and d (0 <= a,b,c,d <= 9) where (a + b + c + d) will be equal to n. Go to the editor

**Input:**

n (1 <= n <= 50)

Input the number(n): 15

Number of combinations: 592

[Click me to see the sample solution](#)

**38.** Write a Python program to print the number of prime numbers which are less than or equal to a given integer. Go to the editor

**Input:**

n (1 <= n <= 999,999)

Input the number(n): 35

Number of prime numbers which are less than or equal to n.: 11

[Click me to see the sample solution](#)

**39.** Write a program to compute the radius and the central coordinate (x, y) of a circle which is constructed by three given points on the plane surface. Go to the editor

**Input:**

x1, y1, x2, y2, x3, y3 separated by a single space.

Input three coordinate of the circle:

9 3 6 8 3 6

Radius of the said circle:

3.358

Central coordinate (x, y) of the circle:

6.071 4.643

[Click me to see the sample solution](#)

**40.** Write a Python program to check whether a point (x,y) is in a triangle or not. There is a triangle formed by three points. Go to the editor

**Input:**

x1,y1,x2,y2,x3,y3,xp,yp separated by a single space.

Input three coordinate of the circle:

9 3 6 8 3 6

Radius of the said circle:

3.358

Central coordinate (x, y) of the circle:

6.071 4.643

[Click me to see the sample solution](#)

- 41.** Write a Python program to compute and print sum of two given integers (more than or equal to zero). If given integers or the sum have more than 80 digits, print "overflow". Go to the editor

Input first integer:

25

Input second integer:

22

Sum of the two integers: 47

[Click me to see the sample solution](#)

- 42.** Write a Python program that accepts six numbers as input and sorts them in descending order. Go to the editor

**Input:**

Input consists of six numbers n1, n2, n3, n4, n5, n6 (-100000 <= n1, n2, n3, n4, n5, n6 <= 100000). The six numbers are separated by a space.

Input six integers:

15 30 25 14 35 40

After sorting the said integers:

40 35 30 25 15 14

[Click me to see the sample solution](#)

- 43.** Write a Python program to test whether two lines PQ and RS are parallel. The four points are P(x1, y1), Q(x2, y2), R(x3, y3), S(x4, y4). Go to the editor

**Input:**

x1,y1,x2,y2,x3,y3,xp,yp separated by a single space

Input x1,y1,x2,y2,x3,y3,xp,yp:

2 5 6 4 8 3 9 7

PQ and RS are not parallel

[Click me to see the sample solution](#)

**44.** Write a Python program to find the maximum sum of a contiguous subsequence from a given sequence of numbers  $a_1, a_2, a_3, \dots, a_n$ . A subsequence of one element is also a continuous subsequence. Go to the editor

**Input:**

You can assume that  $1 \leq n \leq 5000$  and  $-100000 \leq a_i \leq 100000$ .

Input numbers are separated by a space.

Input 0 to exit.

Input number of sequence of numbers you want to input (0 to exit):

3

Input numbers:

2

4

6

Maximum sum of the said contiguous subsequence: 12

Input number of sequence of numbers you want to input (0 to exit):

0

[Click me to see the sample solution](#)

**45.** There are two circles C1 with radius r1, central coordinate  $(x_1, y_1)$  and C2 with radius r2 and central coordinate  $(x_2, y_2)$ . Go to the editor

Write a Python program to test the followings -

- "C2 is in C1" if C2 is in C1
- "C1 is in C2" if C1 is in C2
- "Circumference of C1 and C2 intersect" if circumference of C1 and C2 intersect, and
- "C1 and C2 do not overlap" if C1 and C2 do not overlap.

**Input:**

Input numbers (real numbers) are separated by a space.

Input  $x_1, y_1, r_1, x_2, y_2, r_2$ :

5 6 4 8 7 9

C1 is in C2

[Click me to see the sample solution](#)

**46.** Write a Python program to that reads a date (from 2016/1/1 to 2016/12/31) and prints the day of the date. Jan. 1, 2016, is Friday. Note that 2016 is a leap year. Go to the editor

**Input:**

Two integers m and d separated by a single space in a line, m ,d represent the month and the

day.

Input month and date (separated by a single space):

5 15

Name of the date: Sunday

[Click me to see the sample solution](#)

**47.** Write a Python program which reads a text (only alphabetical characters and spaces.) and prints two words. The first one is the word which is arise most frequently in the text. The second one is the word which has the maximum number of letters. Go to the editor

Note: A word is a sequence of letters which is separated by the spaces.**Input:**

A text is given in a line with following condition:

- a. The number of letters in the text is less than or equal to 1000.
- b. The number of letters in a word is less than or equal to 32.
- c. There is only one word which is arise most frequently in given text.
- d. There is only one word which has the maximum number of letters in given text.

Input text: Thank you for your comment and your participation.

Output: your participation.

[Click me to see the sample solution](#)

**48.** Write a Python program that reads n digits (given) chosen from 0 to 9 and prints the number of combinations where the sum of the digits equals to another given number (s). Do not use the same digits in a combination. Go to the editor

**Input:**

Two integers as number of combinations and their sum by a single space in a line. Input 0 0 to exit.

Input number of combinations and sum, input 0 0 to exit:

5 6

2 4

0 0

2

[Click me to see the sample solution](#)

**49.** Write a Python program which reads the two adjoined sides and the diagonal of a parallelogram and check whether the parallelogram is a rectangle or a rhombus. Go to the editor

According to Wikipedia-

parallelograms: In Euclidean geometry, a parallelogram is a simple (non-self-intersecting) quadrilateral with two pairs of parallel sides. The opposite or facing sides of a parallelogram

are of equal length and the opposite angles of a parallelogram are of equal measure.

rectangles: In Euclidean plane geometry, a rectangle is a quadrilateral with four right angles. It can also be defined as an equiangular quadrilateral, since equiangular means that all of its angles are equal ( $360^\circ/4 = 90^\circ$ ). It can also be defined as a parallelogram containing a right angle.

rhombus: In plane Euclidean geometry, a rhombus (plural rhombi or rhombuses) is a simple (non-self-intersecting) quadrilateral whose four sides all have the same length. Another name is equilateral quadrilateral, since equilateral means that all of its sides are equal in length. The rhombus is often called a diamond, after the diamonds suit in playing cards which resembles the projection of an octahedral diamond, or a lozenge, though the former sometimes refers specifically to a rhombus with a  $60^\circ$  angle, and the latter sometimes refers specifically to a rhombus with a  $45^\circ$  angle.

**Input:**

Two adjoined sides and the diagonal.

$1 \leq ai, bi, ci \leq 1000, ai + bi > ci$

Input two adjoined sides and the diagonal of a parallelogram (comma separated):

3,4,5

This is a rectangle.

[Click me to see the sample solution](#)

**50.** Write a Python program to replace a string "Python" with "Java" and "Java" with "Python" in a given string. Go to the editor

**Input:**

English letters (including single byte alphanumeric characters, blanks, symbols) are given on one line. The length of the input character string is 1000 or less.

Input a text with two words 'Python' and 'Java'

Python is popular than Java

Java is popular than Python

[Click me to see the sample solution](#)

**51.** Write a Python program to find the difference between the largest integer and the smallest integer which are created by 8 numbers from 0 to 9. The number that can be rearranged shall start with 0 as in 00135668. Go to the editor

**Input:**

Input an integer created by 8 numbers from 0 to 9.:

2345

Difference between the largest and the smallest integer from the given integer:

3087

[Click me to see the sample solution](#)

**52.** Write a Python program to compute the sum of first n given prime numbers. Go to the editor

**Input:**

n ( n <= 10000). Input 0 to exit the program.

Input a number (n<=10000) to compute the sum:(0 to exit)

25

Sum of first 25 prime numbers:

1060

[Click me to see the sample solution](#)

**53.** Write a Python program that accept an even number (>=4, Goldbach number) from the user and create a combinations that express the given number as a sum of two prime numbers.

Print the number of combinations. Go to the editor

Goldbach number: A Goldbach number is a positive even integer that can be expressed as the sum of two odd primes.[4] Since four is the only even number greater than two that requires the even prime 2 in order to be written as the sum of two primes, another form of the statement of Goldbach's conjecture is that all even integers greater than 4 are Goldbach numbers.

The expression of a given even number as a sum of two primes is called a Goldbach partition of that number. The following are examples of Goldbach partitions for some even numbers:

6 = 3 + 3

8 = 3 + 5

10 = 3 + 7 = 5 + 5

12 = 7 + 5

...

100 = 3 + 97 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59 = 47 + 53

Input an even number (0 to exit):

100

Number of combinations:

6

[Click me to see the sample solution](#)

**54.** if you draw a straight line on a plane, the plane is divided into two regions. For example, if you pull two straight lines in parallel, you get three areas, and if you draw vertically one to the other you get 4 areas.

Write a Python program to create maximum number of regions obtained by drawing n given straight lines. Go to the editor

**Input:**

(1 <= n <= 10,000)

Input number of straight lines (0 to exit):

5

Number of regions:

16

[Click me to see the sample solution](#)

**55.** There are four different points on a plane, P( $x_p, y_p$ ), Q( $x_q, y_q$ ), R( $x_r, y_r$ ) and S( $x_s, y_s$ ). Write a Python program to test AB and CD are orthogonal or not. Go to the editor

**Input:**

$x_p, y_p, x_q, y_q, x_r, y_r, x_s$  and  $y_s$  are -100 to 100 respectively and each value can be up to 5 digits after the decimal point It is given as a real number including the number of. Output:

Output AB and CD are not orthogonal! or AB and CD are orthogonal!.

[Click me to see the sample solution](#)

**56.** Write a Python program to sum of all numerical values (positive integers) embedded in a sentence. Go to the editor

**Input:**

Sentences with positive integers are given over multiple lines. Each line is a character string containing one-byte alphanumeric characters, symbols, spaces, or an empty line. However the input is 80 characters or less per line and the sum is 10,000 or less.

Input some text and numeric values ( to exit):

Sum of the numeric values: 80

None

Input some text and numeric values ( to exit):

Sum of the numeric values: 17

None

Input some text and numeric values ( to exit):

Sum of the numeric values: 10

None

[Click me to see the sample solution](#)

**57.** There are 10 vertical and horizontal squares on a plane. Each square is painted blue and green. Blue represents the sea, and green represents the land. When two green squares are in contact with the top and bottom, or right and left, they are said to be ground. The area created by only one green square is called "island". For example, there are five islands in the figure below.

Write a Python program to read the mass data and find the number of islands. Go to the editor

**Input:**

Input 10 rows of 10 numbers representing green squares (island) as 1 and blue squares (sea) as zeros

1100000111

1000000111

0000000111

0010001000

0000011100

0000111110

0001111111

1000111110

1100011100

1110001000

Number of islands:

5

[Click me to see the sample solution](#)

**58.** When character are consecutive in a string , it is possible to shorten the character string by replacing the character with a certain rule. For example, in the case of the character string YYYYY, if it is expressed as # 5 Y, it is compressed by one character.

Write a Python program to restore the original string by entering the compressed string with this rule. However, the # character does not appear in the restored character string. Go to the editor

Note: The original sentences are uppercase letters, lowercase letters, numbers, symbols, less than 100 letters, and consecutive letters are not more than 9 letters.

**Input:**

The restored character string for each character on one line.

Original text: XY#6Z1#4023

XYZZZZZ1000023

Original text: #39+1=1#30

999+1=1000

[Click me to see the sample solution](#)

**59.** A convex polygon is a simple polygon in which no line segment between two points on the boundary ever goes outside the polygon. Equivalently, it is a simple polygon whose interior is a convex set. In a convex polygon, all interior angles are less than or equal to 180 degrees, while in a strictly convex polygon all interior angles are strictly less than 180 degrees.

Write a Python program that compute the area of the polygon . The vertices have the names vertex 1, vertex 2, vertex 3, ... vertex n according to the order of edge connections Go to the editor

Note: The original sentences are uppercase letters, lowercase letters, numbers, symbols, less than 100 letters, and consecutive letters are not more than 9 letters.

**Input:**

Input is given in the following format.

x1 , y1

x2 , y2

:

xn , yn

xi , yi are real numbers representing the x and y coordinates of vertex i , respectively.

Input the coordinates (ctrl+d to exit):

1.0, 0.0

0.0, 0.0

1.0, 1.0

2.0, 0.0

-1.0, 1.0

Area of the polygon;

1.50000000.

[Click me to see the sample solution](#)

**60.** Internet search engine giant, such as Google accepts web pages around the world and classify them, creating a huge database. The search engines also analyze the search keywords entered by the user and create inquiries for database search. In both cases, complicated processing is carried out in order to realize efficient retrieval, but basics are all cutting out words from sentences.

Write a Python program to cut out words of 3 to 6 characters length from a given sentence not more than 1024 characters. Go to the editor

**Input:**

English sentences consisting of delimiters and alphanumeric characters are given on one line.

Input a sentence (1024 characters. max.)

The quick brown fox

3 to 6 characters length of words:

The quick brown fox

[Click me to see the sample solution](#)

**61.** Arrange integers (0 to 99) as narrow hilltop, as illustrated in Figure 1. Reading such data representing huge, when starting from the top and proceeding according to the next rule to the bottom. Write a Python program that compute the maximum value of the sum of the passing integers. Go to the editor

**Input:**

A series of integers separated by commas are given in diamonds. No spaces are included in each line. The input example corresponds to Figure 1. The number of lines of data is less than 100 lines.

**Output:**

The maximum value of the sum of integers passing according to the rule on one line.

Input the numbers (ctrl+d to exit):

8

4, 9

9, 2, 1

3, 8, 5, 5

5, 6, 3, 7, 6

3, 8, 5, 5

9, 2, 1

4, 9

8

Maximum value of the sum of integers passing according to the rule on one line.

64

[Click me to see the sample solution](#)

**62.** Write a Python program to find the number of combinations that satisfy  $p + q + r + s = n$  where  $n$  is a given number  $\leq 4000$  and  $p, q, r, s$  in the range of 0 to 1000. Go to the editor

Input a positive integer: (ctrl+d to exit)

252

Number of combinations of a,b,c,d: 2731135

[Click me to see the sample solution](#)

**63.** Write a Python program which adds up columns and rows of given table as shown in the specified figure. Go to the editor

Input number of rows/columns (0 to exit)

4

Input cell value:

25 69 51 26

68 35 29 54

54 57 45 63

61 68 47 59

Result:

25 69 51 26 171

68 35 29 54 186

54 57 45 63 219

61 68 47 59 235

208 229 172 202 811

Input number of rows/columns (0 to exit)

[Click me to see the sample solution](#)

**64.** Given a list of numbers and a number k, write a Python program to check whether the sum of any two numbers from the list is equal to k or not. Go to the editor

For example, given [1, 5, 11, 5] and k = 16, return true since  $11 + 5$  is 16.

Sample Input:

([12, 5, 0, 5], 10)

([20, 20, 4, 5], 40)

([1, -1], 0)

([1, 1, 0], 0)

Sample Output:

True

True

True

False

[Click me to see the sample solution](#)

**65.** In mathematics, a subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. For example, the sequence (A,B,D) is a subsequence of (A,B,C,D,E,F) obtained after removal of elements C, E, and F. The relation of one sequence being the subsequence of another is a preorder.

The subsequence should not be confused with substring (A,B,C,D) which can be derived from the above string (A,B,C,D,E,F) by deleting substring (E,F). The substring is a refinement of the subsequence.

The list of all subsequences for the word "apple" would be "a", "ap", "al", "ae", "app", "apl", "ape", "ale", "appl", "appe", "aple", "apple", "p", "pp", "pl", "pe", "ppl", "ppe", "ple", "pple", "l", "le", "e", "".

Write a Python program to find the longest word in set of words which is a subsequence of a given string. Go to the editor

Sample Input:

("Green", {"Gn", "Gren", "ree", "en"})

("pythonexercises", {"py", "ex", "exercises"})

Sample Output:

Gren

exercises

[Click me to see the sample solution](#)

**66.** From Wikipedia, the free encyclopaedia:

A happy number is defined by the following process:

Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a

cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers.

Write a Python program to check whether a number is "happy" or not. Go to the editor

Sample Input:

(7)

(932)

(6)

Sample Output:

True

True

False

[Click me to see the sample solution](#)

**67. From Wikipedia,**

A happy number is defined by the following process:

Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers.

Write a Python program to find and print the first 10 happy numbers. Go to the editor

Sample Input:

[:10]

Sample Output:

[1, 7, 10, 13, 19, 23, 28, 31, 32, 44]

[Click me to see the sample solution](#)

**68. Write a Python program to count the number of prime numbers less than a given non-negative number. Go to the editor**

Sample Input:

(10)

(100)

Sample Output:

4

25

[Click me to see the sample solution](#)

**69. In abstract algebra, a group isomorphism is a function between two groups that sets up a one-to-one correspondence between the elements of the groups in a way that respects the given group operations. If there exists an isomorphism between two groups, then the groups are**

called isomorphic.

Two strings are isomorphic if the characters in string A can be replaced to get string B

Given "foo", "bar", return false.

Given "paper", "title", return true.

Write a Python program to check if two given strings are isomorphic to each other or not. Go to the editor

Sample Input:

```
("foo", "bar")
("bar", "foo")
("paper", "title")
("title", "paper")
("apple", "orange")
("aa", "ab")
("ab", "aa")
```

Sample Output:

False

False

True

True

False

False

False

[Click me to see the sample solution](#)

**70.** Write a Python program to find the longest common prefix string amongst a given array of strings. Return false If there is no common prefix.

For Example, longest common prefix of "abcdefg" and "abcefgh" is "abc". Go to the editor

Sample Input:

```
["abcdefg","abcefgh"]
["w3r","w3resource"]
["Python","PHP", "Perl"]
["Python","PHP", "Java"]
```

Sample Output:

abc

w3r

P

[Click me to see the sample solution](#)

**71.** Write a Python program to reverse only the vowels of a given string. Go to the editor

Sample Input:

("w3resource")

("Python")

("Perl")

("USA")

Sample Output:

w3resorce

Python

Perl

ASU

[Click me to see the sample solution](#)

**72.** Write a Python program to check whether a given integer is a palindrome or not. Go to the editor

Note: An integer is a palindrome when it reads the same backward as forward. Negative numbers are not palindromic.

Sample Input:

(100)

(252)

(-838)

Sample Output:

False

True

False

[Click me to see the sample solution](#)

**73.** Write a Python program to remove the duplicate elements of a given array of numbers such that each element appear only once and return the new length of the given array. Go to the editor

Sample Input:

[0,0,1,1,2,2,3,3,4,4,4]

[1, 2, 2, 3, 4, 4]

Sample Output:

5

4

[Click me to see the sample solution](#)

**74.** Write a Python program to calculate the maximum profit from selling and buying values of stock. An array of numbers represent the stock prices in chronological order. Go to the editor  
For example, given [8, 10, 7, 5, 7, 15], the function will return 10, since the buying value of the stock is 5 dollars and sell value is 15 dollars.

Sample Input:

([8, 10, 7, 5, 7, 15])

([1, 2, 8, 1])

([])

Sample Output:

10

7

0

[Click me to see the sample solution](#)

**75.** Write a Python program to remove all instances of a given value from a given array of integers and find the length of the new array. Go to the editor

Sample Input:

([1, 2, 3, 4, 5, 6, 7, 5], 5)

([10, 10, 10, 10, 10], 10)

([10, 10, 10, 10, 10], 20)

([], 1)

Sample Output:

6

0

5

0

[Click me to see the sample solution](#)

**76.** Write a Python program to find the starting and ending position of a given value in a given array of integers, sorted in ascending order. Go to the editor

If the target is not found in the array, return [0, 0].

Input: [5, 7, 7, 8, 8, 8] target value = 8

Output: [0, 5]

Input: [1, 3, 6, 9, 13, 14] target value = 4

Output: [0, 0]

[Click me to see the sample solution](#)

**77.** The price of a given stock on each day is stored in an array.

Write a Python program to find the maximum profit in one transaction i.e., buy one and sell one

share of the stock from the given price value of the said array. You cannot sell a stock before you buy one. Go to the editor

Input (Stock price of each day): [224, 236, 247, 258, 259, 225]

Output: 35

Explanation:

236 - 224 = 12

247 - 224 = 23

258 - 224 = 34

259 - 224 = 35

225 - 224 = 1

247 - 236 = 11

258 - 236 = 22

259 - 236 = 23

225 - 236 = -11

258 - 247 = 11

259 - 247 = 12

225 - 247 = -22

259 - 258 = 1

225 - 258 = -33

225 - 259 = -34

[Click me to see the sample solution](#)

**78.** Write a Python program to print a given N by M matrix of numbers line by line in forward > backwards > forward >... order. Go to the editor

Input matrix:

[[1, 2, 3, 4],

[5, 6, 7, 8],

[0, 6, 2, 8],

[2, 3, 0, 2]]

Output:

1

2

3

4

8

7

6

5

0

6  
2  
8  
2  
0  
3  
2

[Click me to see the sample solution](#)

**79.** Write a Python program to compute the largest product of three integers from a given list of integers. Go to the editor

Sample Input:

[-10, -20, 20, 1]  
[-1, -1, 4, 2, 1]  
[1, 2, 3, 4, 5, 6]

Sample Output:

```
1 4000
2 8
3 120
```

[Click me to see the sample solution](#)

**80.** Write a Python program to find the first missing positive integer that does not exist in a given list. Go to the editor

Sample Input:

[2, 3, 7, 6, 8, -1, -10, 15, 16]  
[1, 2, 4, -7, 6, 8, 1, -10, 15]  
[1, 2, 3, 4, 5, 6, 7]  
[-2, -3, -1, 1, 2, 3]

Sample Output:

```
1 4
2 3
3 8
4 4
```

[Click me to see the sample solution](#)

**81.** Write a Python program to randomly generate a list with 10 even numbers between 1 and 100 inclusive. Go to the editor

Note: Use `random.sample()` to generate a list of random values.

Sample Input:

(1,100)

Sample Output:

```
[4, 22, 8, 20, 24, 12, 30, 98, 28, 48]
```

[Click me to see the sample solution](#)

**82.** Write a Python program to calculate the median from a list of numbers. Go to the editor

Sample Input:

[1,2,3,4,5]

[1,2,3,4,5,6]

[6,1,2,4,5,3]

[1.0,2.11,3.3,4.2,5.22,6.55]

[1.0,2.11,3.3,4.2,5.22]

[2.0,12.11,22.3,24.12,55.22]

Sample Output:

3

3.5

3.5

3.75

3.3

22.3

[Click me to see the sample solution](#)

**83.** Write a Python program to test whether a given number is symmetrical or not. Go to the editor

A number is symmetrical when it is equal of its reverse.

Sample Input:

(121)

(0)

(122)

(990099)

Sample Output:

True

True

False

True

[Click me to see the sample solution](#)

**84.** Write a Python program that accept a list of numbers and create a list to store the count of negative number in first element and store the sum of positive numbers in second element. Go to the editor

Sample Input:

[1, 2, 3, 4, 5]

[-1, -2, -3, -4, -5]

[1, 2, 3, -4, -5]

[1, 2, -3, -4, -5]

Sample Output:

[0, 15]

[5, 0]

[2, 6]

[3, 3]

[Click me to see the sample solution](#)

**85.** From Wikipedia:

An isogram (also known as a "nonpattern word") is a logological term for a word or phrase without a repeating letter. It is also used by some people to mean a word or phrase in which each letter appears the same number of times, not necessarily just once. Conveniently, the word itself is an isogram in both senses of the word, making it autological.

Write a Python program to check whether a given string is an "isogram" or not. Go to the editor

Sample Input:

("w3resource")

("w3r")

("Python")

("Java")

Sample Output:

False

True

True

False

[Click me to see the sample solution](#)

**86.** Write a Python program to count the number of equal numbers from three given integers.

[Go to the editor](#)

Sample Input:

(1, 1, 1)

(1, 2, 2)

(-1, -2, -3)

(-1, -1, -1)

Sample Output:

3

2

0

3

[Click me to see the sample solution](#)

**87.** Write a Python program to check whether a given employee code is exactly 8 digits or 12 digits. Return True if the employee code is valid and False if it's not. [Go to the editor](#)

Sample Input:

('12345678')

('1234567j')

('12345678j')

('123456789123')

('123456abcdef')

Sample Output:

True

False

False

True

False

[Click me to see the sample solution](#)

**88.** Write a Python program that accept two strings and test if the letters in the second string are present in the first string. [Go to the editor](#)

Sample Input:

["python", "yptn"]

["python", "yptns"]

["python", "yptnon"]

["123456", "01234"]

["123456", "1234"]

Sample Output:

True

False

True

False

True

[Click me to see the sample solution](#)

**89.** Write a Python program to compute the sum of the three lowest positive numbers from a given list of numbers. Go to the editor

Sample Input:

[10, 20, 30, 40, 50, 60, 7]

[1, 2, 3, 4, 5]

[0, 1, 2, 3, 4, 5]

Sample Output:

37

6

6

[Click me to see the sample solution](#)

**90.** Write a Python program to replace all but the last five characters of a given string into "\*" and returns the new masked string. Go to the editor

Sample Input:

("kdi39323swe")

("12345abcdef")

("12345")

Sample Output:

\*\*\*\*\*23swe

\*\*\*\*\*bcdef

12345

[Click me to see the sample solution](#)

**91.** Write a Python program to count the number of arguments in a given function. Go to the editor

Sample Input:

()

(1)

(1, 2)  
(1, 2, 3)  
(1, 2, 3, 4)  
[1, 2, 3, 4]

Sample Output:

0  
1  
2  
3  
4  
1

[Click me to see the sample solution](#)

**92.** Write a Python program to compute cumulative sum of numbers of a given list. Go to the editor

Note: Cumulative sum = sum of itself + all previous numbers in the said list.

Sample Input:

[10, 20, 30, 40, 50, 60, 7]  
[1, 2, 3, 4, 5]  
[0, 1, 2, 3, 4, 5]

Sample Output:

[10, 30, 60, 100, 150, 210, 217]  
[1, 3, 6, 10, 15]  
[0, 1, 3, 6, 10, 15]

[Click me to see the sample solution](#)

**93.** Write a Python program to find the middle character(s) of a given string. If the length of the string is odd return the middle character and return the middle two characters if the string length is even. Go to the editor

Sample Input:

("Python")  
("PHP")  
("Java")

Sample Output:

th  
H  
av

[Click me to see the sample solution](#)

**94.** Write a Python program to find the largest product of the pair of adjacent elements from a given list of integers. Go to the editor

Sample Input:

[1,2,3,4,5,6]

[1,2,3,4,5]

[2,3]

Sample Output:

30

20

6

[Click me to see the sample solution](#)

**95.** Write a Python program to check whether every even index contains an even number and every odd index contains odd number of a given list. Go to the editor

Sample Input:

[2, 1, 4, 3, 6, 7, 6, 3]

[2, 1, 4, 3, 6, 7, 6, 4]

[4, 1, 2]

Sample Output:

True

False

True

[Click me to see the sample solution](#)

**96.** Write a Python program to check whether a given number is a narcissistic number or not.

Go to the editor

If you are a reader of Greek mythology, then you are probably familiar with Narcissus. He was a hunter of exceptional beauty that he died because he was unable to leave a pool after falling in love with his own reflection. That's why I keep myself away from pools these days (kidding).

In mathematics, he has kins by the name of narcissistic numbers - numbers that can't get enough of themselves. In particular, they are numbers that are the sum of their digits when raised to the power of the number of digits.

For example, 371 is a narcissistic number; it has three digits, and if we cube each digits  $3^3 + 7^3 + 1^3$  the sum is 371. Other 3-digit narcissistic numbers are

$153 = 1^3 + 5^3 + 3^3$

$370 = 3^3 + 7^3 + 0^3$

$407 = 4^3 + 0^3 + 7^3$

There are also 4-digit narcissistic numbers, some of which are 1634, 8208, 9474 since

1634 = 14+64+34+44

8208 = 84+24+04+84

9474 = 94+44+74+44

It has been proven that there are only 88 narcissistic numbers (in the decimal system) and that the largest of which is

115,132,219,018,763,992,565,095,597,973,971,522,401

has 39 digits. Ref.: //<https://bit.ly/2qNYxo2>

Sample Input:

(153)

(370)

(407)

(409)

(1634)

(8208)

(9474)

(9475)

Sample Output:

True

True

True

False

True

True

True

False

[Click me to see the sample solution](#)

**97.** Write a Python program to find the highest and lowest number from a given string of space separated integers. Go to the editor

Sample Input:

("1 4 5 77 9 0")

("-1 -4 -5 -77 -9 0")

("0 0")

Sample Output:

(77, 0)

(0, -77)

(0, 0)

[Click me to see the sample solution](#)

**98.** Write a Python program to check whether a sequence of numbers has an increasing trend or not. Go to the editor

Sample Input:

[1,2,3,4]

[1,2,5,3,4]

[-1,-2,-3,-4]

[-4,-3,-2,-1]

[1,2,3,4,0]

Sample Output:

True

False

False

True

False

[Click me to see the sample solution](#)

**99.** Write a Python program to find the position of the second occurrence of a given string in another given string. If there is no such string return -1. Go to the editor

Sample Input:

("The quick brown fox jumps over the lazy dog", "the")

("the quick brown fox jumps over the lazy dog", "the")

Sample Output:

-1

31

[Click me to see the sample solution](#)

**100.** Write a Python program to compute the sum of all items of a given array of integers where each integer is multiplied by its index. Return 0 if there is no number. Go to the editor

Sample Input:

[1,2,3,4]

[-1,-2,-3,-4]

[]

Sample Output:

20

-20

0

[Click me to see the sample solution](#)

**101.** Write a Python program to find the name of the oldest student from a given dictionary containing the names and ages of a group of students. Go to the editor

Sample Input:

```
{"Bernita Ahner": 12, "Kristie Marsico": 11, "Sara Pardee": 14, "Fallon Fabiano": 11, "Nidia Dominique": 15}
 {"Nilda Woodside": 12, "Jackelyn Pineda": 12.2, "Sofia Park": 12.4, "Joannie Archibald": 12.6,
 "Becki Saunder": 12.7}
```

Sample Output:

Nidia Dominique

Becki Saunder

[Click me to see the sample solution](#)

**102.** Write a Python program to create a new string with no duplicate consecutive letters from a given string. Go to the editor

Sample Input:

```
("PPYYYTTHON")
("PPyythonnn")
("Java")
("PPPHHHPPP")
```

Sample Output:

PYTHON

Python

Java

PHP

[Click me to see the sample solution](#)

**103.** Write a Python program to check whether two given lines are parallel or not. Go to the editor

Note: Parallel lines are two or more lines that never intersect. Parallel Lines are like railroad tracks that never intersect.

The General Form of the equation of a straight line is:  $ax + by = c$

The said straight line is represented in a list as [a, b, c]

Example of two parallel lines:

$x + 4y = 10$  and  $x + 4y = 14$

Sample Input:

```
([2,3,4], [2,3,8])
([2,3,4], [4,-3,8])
```

Sample Output:

True

False

[Click me to see the sample solution](#)

**104.** Write a Python program to find the lucky number(s) in a given matrix. Go to the editor

Sample Input:

Original matrix: [[1, 2], [2, 3]]

Lucky number(s) in the said matrix: [2]

Original matrix: [[1, 2, 3], [3, 4, 5]]

Lucky number(s) in the said matrix: [3]

Original matrix: [[7, 5, 6], [3, 4, 4], [6, 5, 7]]

Lucky number(s) in the said matrix: [5]

[Click me to see the sample solution](#)

**105.** Write a Python program to check whether a given sequence is linear, quadratic or cubic. Go to the editor

Sequences are sets of numbers that are connected in some way.

Linear sequence:

A number pattern which increases or decreases by the same amount each time is called a linear sequence. The amount it increases or decreases by is known as the common difference.

Quadratic sequence:

In quadratic sequence, the difference between each term increases, or decreases, at a constant rate.

Cubic sequence:

Sequences where the 3rd difference are known as cubic sequence.

Sample Input:

[0,2,4,6,8,10]

[1,4,9,16,25]

[0,12,10,0,-12,-20]

[1,2,3,4,5]

Sample Output:

Linear Sequence

Quadratic Sequence

Cubic Sequence

Linear Sequence

[Click me to see the sample solution](#)

**106.** Write a Python program to test whether a given integer is pandigital number or not. Go to the editor

From Wikipedia,

In mathematics, a pandigital number is an integer that in a given base has among its significant digits each digit used in the base at least once.

For example,

12233344445555566666677777788888889999999990 is a pandigital number in base 10.

The first few pandigital base 10 numbers are given by:

1023456789, 1023456798, 1023456879, 1023456897, 1023456978, 1023456987,

1023457689

Sample Input:

(1023456897)

(1023456798)

(1023457689)

(1023456789)

(102345679)

Sample Output:

True

True

True

True

False

[Click me to see the sample solution](#)

**107.** Write a Python program to check whether a given number is Oddish or Evenish. Go to the editor

A number is called "Oddish" if the sum of all of its digits is odd, and a number is called "Evenish" if the sum of all of its digits is even.

Sample Input:

(120)

(321)

(43)

(4433)

(373)

Sample Output:

Oddish

Evenish

Oddish

Evenish

Oddish

[Click me to see the sample solution](#)

**108.** Write a Python program that takes three integers and check whether the last digit of first number \* the last digit of second number = the last digit of third number. Go to the editor

Sample Input:

(12, 22, 44)  
(145, 122, 1010)  
(0, 22, 40)  
(1, 22, 40)  
(145, 122, 101)

Sample Output:

True  
True  
True  
False  
False

[Click me to see the sample solution](#)

**109.** Write a Python program find the indices of all occurrences of a given item in a given list.

Go to the editor

Sample Input:

([1,2,3,4,5,2], 2)  
([3,1,2,3,4,5,6,3,3], 3)  
([1,2,3,-4,5,2,-4], -4)

Sample Output:

[1, 5]  
[0, 3, 7, 8]  
[3, 6]

[Click me to see the sample solution](#)

**110.** Write a Python program to remove two duplicate numbers from a given number of list. Go

to the editor

Sample Input:

([1,2,3,2,3,4,5])  
([1,2,3,2,4,5])  
([1,2,3,4,5])

Sample Output:

[1, 4, 5]  
[1, 3, 4, 5]  
[1, 2, 3, 4, 5]

[Click me to see the sample solution](#)

**111.** Write a Python program to check whether two given circles (given center (x,y) and radius) are intersecting. Return true for intersecting otherwise false. Go to the editor

Sample Input:

([1,2, 4], [1,2, 8])

([0,0, 2], [10,10, 5])

Sample Output:

True

False

[Click me to see the sample solution](#)

**112.** Write a Python program to compute the digit distance between two integers. Go to the editor

The digit distance between two numbers is the absolute value of the difference of those numbers.

For example, the distance between 3 and -3 on the number line given by the  $|3 - (-3)| = |3 + 3| = 6$  units

Digit distance of 123 and 256 is

Since  $|1 - 2| + |2 - 5| + |3 - 6| = 1 + 3 + 3 = 7$

Sample Input:

(123, 256)

(23, 56)

(1, 2)

(24232, 45645)

Sample Output:

7

6

1

11

[Click me to see the sample solution](#)

**113.** Write a Python program to reverse all the words which have even length. Go to the editor

Sample Input:

("The quick brown fox jumps over the lazy dog")

("Python Exercises")

Sample Output:

The quick brown fox jumps revo the yzal dog

nohtyP Exercises

[Click me to see the sample solution](#)

**114.** Write a Python program to print letters from the English alphabet from a-z and A-Z. Go to the editor

Sample Input:

("Alphabet from a-z:")

("\\nAlphabet from A-Z:")

Sample Output:

Alphabet from a-z:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Alphabet from A-Z:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

[Click me to see the sample solution](#)

**115.** Write a Python program to generate and prints a list of numbers from 1 to 10. Go to the editor

Sample Input:

range(1,10)

Sample Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

[1', '2', '3', '4', '5', '6', '7', '8', '9']

[Click me to see the sample solution](#)

**116.** Write a Python program to identify nonprime numbers between 1 to 100 (integers). Print the nonprime numbers. Go to the editor

Sample Input:

range(1, 101)

Sample Output:

Nonprime numbers between 1 to 100:

4

6

8

9

10

..

96

98

99

100

[Click me to see the sample solution](#)

**117.** Write a Python program to make a request to a web page, and test the status code, also display the html code of the specified web page. Go to the editor

Sample Output:

Web page status: <Response [200]>

HTML code of the above web page:

```
<!doctype html>
<html>
<head>
<title>Example Domain</title>
<meta charset="utf-8" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
</head>
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents. You may use this
domain in literature without prior coordination or asking for permission.</p>
<p>More]
(https://www.iana.org/domains/example">More]
(https://www.iana.org/domains/example">More]28https://www.iana.org/domains/example">
More)\) information...</p>
</div>
</body>
</html>
```

Click me to see the sample solution

**118.** In multiprocessing, processes are spawned by creating a Process object. Write a Python program to show the individual process IDs (parent process, process id etc.) involved. Go to the editor

Sample Output:

Main line

```
module name: __main__
parent process: 23967
process id: 27986
function f
module name: __main__
parent process: 27986
```

process id: 27987

hello bob

[Click me to see the sample solution](#)

**119.** Write a Python program to check if two given numbers are Co Prime or not. Return True if two numbers are Co Prime otherwise return false. Go to the editor

Sample Input:

(17, 13)

(17, 21)

(15, 21)

(25, 45)

Sample Output:

True

True

False

False

[Click me to see the sample solution](#)

**120.** Write a Python program to calculate Euclid's totient function of a given integer. Use a primitive method to calculate Euclid's totient function. Go to the editor

Sample Input:

(10)

(15)

(33)

Sample Output:

4

8

20

[Click me to see the sample solution](#)

**121.** Write a Python program to create a coded string from a given string, using specified formula. Go to the editor

Replace all 'P' with '9', 'T' with '0', 'S' with '1', 'H' with '6' and 'A' with '8'

Original string: PHP

Coded string: 969

Original string: JAVASCRIPT

Coded string: J8V81CRI90

[Click me to see the sample solution](#)

**122.** Write a Python program to check if a given string contains only lowercase or uppercase characters. Go to the editor

Original string: PHP

Coded string: True

Original string: javascript

Coded string: True

Original string: JavaScript

Coded string: False

[Click me to see the sample solution](#)

**123.** Write a Python program to remove the first and last elements from a given string. Go to the editor

Original string: PHP

Removing the first and last elements from the said string: H

Original string: Python

Removing the first and last elements from the said string: ytho

Original string: JavaScript

Removing the first and last elements from the said string: avaScrip

[Click me to see the sample solution](#)

**124.** Write a Python program to check if a given string contains two similar consecutive letters.

Go to the editor

Original string: PHP

Check for consecutive similar letters! False

Original string: PHHP

Check for consecutive similar letters! True

Original string: PHPP

Check for consecutive similar letters! True

[Click me to see the sample solution](#)

**125.** Write a Python program to reverse a given string in lower case. Go to the editor

Original string: PHP

Reverse the said string in lower case: php

Original string: JavaScript

Reverse the said string in lower case: tpircsavaj

Original string: PHPP

Reverse the said string in lower case: pphp

[Click me to see the sample solution](#)

**126.** Write a Python program to convert the letters of a given string (same case-upper/lower) into alphabetical order. Go to the editor

Original string: PHP

Convert the letters of the said string into alphabetical order: HPP

Original string: javascript

Convert the letters of the said string into alphabetical order: aacijprstv

Original string: python

Convert the letters of the said string into alphabetical order: hnooty

[Click me to see the sample solution](#)

**127.** Write a Python program to check whether the average value of the elements of a given array of numbers is a whole number or not. Go to the editor

Original array:

1 3 5 7 9

Check the average value of the elements of the said array is a whole number or not: True

Original array:

2 4 2 6 4 8

Check the average value of the elements of the said array is a whole number or not:

False

[Click me to see the sample solution](#)

**128.** Write a Python program to remove all vowels from a given string. Go to the editor

Original string: Python

After removing all the vowels from the said string: Pythn

Original string: C Sharp

After removing all the vowels from the said string: C Shrp

Original string: JavaScript

After removing all the vowels from the said string: JvScrt

[Click me to see the sample solution](#)

**129.** Write a Python program to get the index number of all lower case letters in a given string.

Go to the editor

Original string: Python

Indices of all lower case letters of the said string: [1, 2, 3, 4, 5] Original string: JavaScript

Indices of all lower case letters of the said string: [1, 2, 3, 5, 6, 7, 8, 9] Original string: PHP

Indices of all lower case letters of the said string: []

[Click me to see the sample solution](#)

**130.** Write a Python program to check whether a given month and year contains a Monday 13th. Go to the editor

Month No.: 11 Year: 2022

Check whether the said month and year contains a Monday 13th.: False

Month No.: 6 Year: 2022

Check whether the said month and year contains a Monday 13th.: True

[Click me to see the sample solution](#)

**131.** Write a Python program to count number of zeros and ones in the binary representation of a given integer. Go to the editor

Original number: 12

Number of ones and zeros in the binary representation of the said number: Number of zeros: 2,

Number of ones: 2

Original number: 1234

Number of ones and zeros in the binary representation of the said number: Number of zeros: 6,

Number of ones: 5

[Click me to see the sample solution](#)

**132.** Write a Python program to find all the factors of a given natural number. Go to the editor

Factors:

The factors of a number are the numbers that divide into it exactly. The number 12 has six factors:

1, 2, 3, 4, 6 and 12 If 12 is divided by any of the six factors then the answer will be a whole number. For example:

$12 / 3 = 4$

Original Number: 1

Factors of the said number: {1}

Original Number: 12

Factors of the said number: {1, 2, 3, 4, 6, 12}

Original Number: 100

Factors of the said number: {1, 2, 4, 100, 5, 10, 50, 20, 25}

[Click me to see the sample solution](#)

**133.** Write a Python program to compute the sum of the negative and positive numbers of an array of integers and display the largest sum. Go to the editor

Original array elements: {0, 15, 16, 17, -14, -13, -12, -11, -10, 18, 19, 20}

Largest sum - Positive/Negative numbers of the said array: 105

Original array elements: {0, 3, 4, 5, 9, -22, -44, -11}

Largest sum - Positive/Negative numbers of the said array: -77

[Click me to see the sample solution](#)

**134.** Write a Python program to alternate the case of each letter in a given string and the first letter of the said string must be uppercase. Go to the editor

Original string: Python Exercises

After alternating the case of each letter of the said string: PyThOn ExErCiSeS

Original string: C# is used to develop web apps, desktop apps, mobile apps, games and much more.

After alternating the case of each letter of the said string: C# iS uSeD tO dEvElOp WeB aPpS, dEsKtOp ApPs, MoBiLe ApPs, GaMeS aNd MuCh MoRe.

[Click me to see the sample solution](#)

**135.** Write a Python program to get the Least Common Multiple (LCM) of more than two numbers. Take the numbers from a given list of positive integers. Go to the editor

From Wikipedia,

In arithmetic and number theory, the least common multiple, lowest common multiple, or smallest common multiple of two integers  $a$  and  $b$ , usually denoted by  $\text{lcm}(a, b)$ , is the smallest positive integer that is divisible by both  $a$  and  $b$ . Since division of integers by zero is undefined, this definition has meaning only if  $a$  and  $b$  are both different from zero. However, some authors define  $\text{lcm}(a, 0)$  as 0 for all  $a$ , which is the result of taking the lcm to be the least upper bound in the lattice of divisibility.

Original list elements: [4, 6, 8]

LCM of the numbers of the said array of positive integers: 24

Original list elements: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

LCM of the numbers of the said array of positive integers: 2520

Original list elements: [48, 72, 108]

LCM of the numbers of the said array of positive integers: 432

[Click me to see the sample solution](#)

**136.** Write a Python program to reverse all the words which have odd length. Go to the editor

Original string: The quick brown fox jumps over the lazy dog

Reverse all the words of the said string which have odd length: ehT kciuq nworb xof spmuj over eht lazy god

Original string: Python Exercises

Reverse all the words of the said string which have odd length: Python sesicrexE

[Click me to see the sample solution](#)

**137.** Write a Python program to find the longest common ending between two given strings. Go to the editor

Original strings: running ruminating

Common ending between said two strings: ing

Original strings: thisisatest testing123testing

Common ending between said two strings:

[Click me to see the sample solution](#)

**138.** Write a Python program to reverse the binary representation of an given integer and convert the reversed binary number into an integer. Go to the editor

Original number: 13

Reverse the binary representation of the said integer and convert it into an integer: 11

Original number: 145

Reverse the binary representation of the said integer and convert it into an integer: 137

Original number: 1342

Reverse the binary representation of the said integer and convert it into an integer: 997

[Click me to see the sample solution](#)

**139.** Write a Python program to find the closest palindrome number of a given integer. If there are two palindrome numbers in absolute distance return the smaller number. Go to the editor

Original number: 120

Closest Palindrome number of the said number: 121

Original number: 321

Closest Palindrome number of the said number: 323

Original number: 43

Closest Palindrome number of the said number: 44

Original number: 1234

Closest Palindrome number of the said number: 1221

[Click me to see the sample solution](#)

**140.** Write a Python program to convert all items in a given list to float values. Go to the editor

Original list:

```
[0.49, '0.54', '0.54', '0.54', '0.54', '0.54', '0.55', '0.54', '0.54', '0.54', '0.54', '0.55', '0.55', '0.55', '0.55', '0.54', '0.55', '0.55', '0.54']
```

List of Floats:

```
[0.49, 0.54, 0.54, 0.54, 0.54, 0.54, 0.55, 0.54, 0.54, 0.54, 0.54, 0.55, 0.55, 0.55, 0.54, 0.55, 0.55, 0.54]
```

[Click me to see the sample solution](#)

**141.** Write a Python program to get the domain name using PTR DNS records from a given IP address. Go to the editor

Domain name using PTR DNS:

dns.google

ec2-13-251-106-90.ap-southeast-1.compute.amazonaws.com

dns.google

ec2-23-23-212-126.compute-1.amazonaws.com

[Click me to see the sample solution](#)

**142.** Write a Python program to check if every consecutive sequence of zeroes is followed by a consecutive sequence of ones of same length in a given string. Return True/False. Go to the editor

Original sequence: 01010101

Check if every consecutive sequence of zeroes is followed by a consecutive sequence of ones in the said string:

True

Original sequence: 00

Check if every consecutive sequence of zeroes is followed by a consecutive sequence of ones in the said string:

False

Original sequence: 000111000111

Check if every consecutive sequence of zeroes is followed by a consecutive sequence of ones in the said string:

True

Original sequence: 00011100011

Check if every consecutive sequence of zeroes is followed by a consecutive sequence of ones in the said string:

False

[Click me to see the sample solution](#)

# Untitled

## Course Content

Before I get started here is the Course Content

- Comments
  - Single-Line
  - Multi-Line
- Print Statements
  - Single-Line
  - Multi-Line
- Data Types
  - Strings
  - Integers
  - Floating Points
  - Booleans
- Variables
  - Proper Variable Names
  - Printing them
  - Changing them
  - Type Function
  - f-strings
  - .format
- String Methods
  - indexing
  - slicing
- Concatenation
  - Printing Using Concatenation
  - Assigning Variables Using Concatenation
  - Type Casting
- Operators
  - Basic Operators
  - Assignment Operators
  - Comparison Operators
- Getting Inputs
  - Storing inputs in a variable
  - Specifying input type

- Practice Problem
- Lists
  - List Methods
- Conditional Statements
  - if
  - else
  - elif
  - pass
  - logic
- Practice Problem
- Loops
  - for Loops
  - while Loops
- Practice Problem
- Escape Codes
- Functions
  - Return
  - Parameters
  - Lambda
  - Scope
    - Global
    - Local

**Let's Get Started!**

---

## Comments

### Single-Line Comments

Comments are blocks of code that are ignored by the computer and can be used to stay stuff about your code. The way to tell your computer that you are entering a comment is with a `#`.

So here's an example:

```
#This is a comment
```

## Multi-Line Comments

You can also have comments that span multiple lines which are done with three single or double quotes to start and 3 to end.

Here's an example:

```
1 #This is a single line comment
2 '''
3 This
4 is
5 a
6 multiline
7 comment
8 '''
```

Comments are mostly used when you want to say something about how your code works, so when you revisit your code you aren't confused on what you were doing. Another common use is when you want to leave some code out of your program and not have it run without completely removing it from your code.

## Printing

The `print()` function prints(or outputs) something to the console.

### Single-Line Printing

Here's an example:

```
1 print("Python Rules")
2 #You can use either Double Quotes or Single Quotes, but using Double Quotes is
3 print('Python Rules')
4 #This works too
```

This would output

```
1 Python Rules
2 Python Rules
```

## Multi-Line Printing

If you want to print text for multiple lines then use 3 single quotes or doubles quotes to start and 3 to end.

Here's an example:

```
1 print("Here's my to do list")
2
3 print("""
4
5 To Do:
6
7 [1] Code In Python
8 [2] Code In Python
9 [3] Code In Python
10 """)
```

The Output:

```
1 Here's My to do list
2
3
4 To Do:
5
6 [1] Code In Python
7 [2] Code In Python
8 [3] Code In Python
```

## Data Types

`str` or `string` is a data type that is a sequence of characters and is surrounded in double or single quotes

Ex: "Python Rocks" , 'Python is Fun'

`int` or `integer` is a any positive or negative number (without decimals)

Ex: 5 , 69 , -568

`float` is any positive or negative number with a decimals

Ex: 69.69 , 5.0 , -15.89

`bool` or `boolean` is a True or False(It can `only` be `True` or `False` ) value that can be used for Logic (Make sure to capatalize it)

---

## Variables

Variables are data types that are assigned to variable names and are used to store information.

### Proper Variable Names

They can use numbers,letters,underscores, but can't start with numbers. No spaces are allowed either(Use underscores instead). You should use variable names that relate to the value being stored, and variable names that are also short.

Here are a few example of valid and good Variable names:

```
1 Apple_Cost = 0.49
2 Apples_Amnt = 5
```

### Printing Variables

You can also print variables by putting them in the parenthesis of the print Statements

```
1 name = "IntellectualGuy"
2 print(name)
```

It would output `IntellectualGuy`

## Reassigning Variables

You can also change the value of the variable by simply just reassigning it

```
1 Pog = False
2 Pog = True
```

You can also switch the data type

```
1 Bannanas = False
2 Bannanas = 3
```

## Type Function

After you change a variable a lot you might want to find out what data type it is so you just use the `type()` function.

```
1 Apples = 5
2 print(type(Apples))
```

It would output `<class 'int'>`, which basically means that the data type of the variable apple is an int or integer.

## F-Strings

F-Strings print out stuff with a variable in between. This will become more useful when you learn about inputs.

Here's an example:

```
1 username = "IntellectualGuy"
2 print(f"Hello {username}")
3 cycles = 100
4 print(f"Hello {username} you have {cycles} cycles")
```

It would output:

```
1 Hello IntellectualGuy
2 Hello IntellectualGuy you have 100 cycles
```

## .format()

Another Way to do that is using the .format method which you use like so

```
1 print("Hello {}".format("IntellectualGuy"))
2
3 #You can insert more than one value
4 print("Hello {} you have {} cycles".format(username,cycles))
5
6 #You can use variables
7 print("How many {} would you like to buy? Each costs ${}.format(fruit, cost))
8
9 #You can also use indexing
10 print("Hello {} you bought {} bannanas".format("IntellectualGuy",3))
```

Output:

```
1 Hello IntellectualGuy
2 Hello IntellectualGuy you have 100 cycles
3 How many bannanas would you like to buy? Each costs $0.75.
4 Hello IntellectualGuy you bought 3 bannanas
```

## String Methods

There are many different string methods that you can use. String methods are basically functions that you can apply to your strings. For example the `print()` function that I explained earlier can print a string to the console.

Something to know before starting string methods is that each character in a string has an index value, starting from 0.

Here's an example

```
1 "H e l l o"
2 0 1 2 3 4
3
4 #This shows in the string hello, what characters are in what index positions
```

Index positions can also be in the negatives like so

```
1 "H e l l o"
2 -5 -4 -3 -2 -1
```

Negative index positions are mainly used for things like getting the last character of a string

## Indexing

Indexing is getting the value of a certain index of a string, and you use it like so

```
1 greeting = "Hello"
2 print(greeting[3])
```

Output: l

You can also use negative indexing to get a character

```
1 greeting = "Hello"
2 print(greeting[-1])
```

Output: o

## Slicing

Slicing is used to get a certain part of a string like so

## Length

Length is used to get the length , the syntax is `len(string)` of a string and is used like so

```
1 greeting = "Hello"
2 print(len(greeting))
```

Output: 5

## Upper

The `.upper()` method is used to uppercase a string like so

```
1 name = "bob"
2 print(name.upper())
```

Output: BOB

## Lower

The `.lower()` method is used to lowercase a string like so

```
1 name = "SAM"
2 print(name.lower())
```

Output: sam

---

## Concatenation

Concatenation is when you add/join 2 or more strings together.

For example:

```
1 User_Type = "Admin"
2 print("You are the" + User_Type)
```

Output: You are the Admin.

You can also assign a variable using Concatenation

For example:

```
1 User_Type = "Guest"
2 Message = "You are a" + User_Type
3 print(Message)
```

Output: You are a Guest

Remember you can Concatenate more than 2 strings together

For Example:

```
1 User_Type = "Admin"
2 Message = "Hello" + User_Type + "What would you like to do?"
3 print(Message)
```

Output: Hello Admin What would you like to do?

## Type Casting

You can't Concatenate strings with integers/floatsBOOLEANS, you would have to cast the type, let's say you wanted to print out a message with a number using Concatenation, then you would use type casting, which lets you change the type of a variable.

Ex:

```
1 Age = 5
2
3 print("Bob you are " + Age + " years old.")
4 #That would produce a Type error, and the way to fix that is to cast the integer
5
6 #Let's do it again without producing an error
7 Age = 5
8 print("Bob you are " + str(Age) + " years old.")
9
10 #The syntax for type casting is the data type you want to convert the variable to followed by (variable)
11 #datatype(variable)
12
13 #Let's try another Example
14 Name = "Bob"
15 print("I know someone who is 5 years old, his name is" + int(Name))
16 #This would produce an error because you can't convert the string into an integer
```

Note: Type casting only temporarily changes the data type of the variable, not permanently, If you wanted to though then you could assign the variable to the casted variable like solution

```
1 age = 5
2 age = str(age)
3 #Now instead of being the integer 5, age is now the string "5"
```

# Operators

## Basic Operators

You can do basic Math with Python.

Addition - You can add Two numbers with a plus sign `+`

Ex: `5 + 5`, `6.9 + 9.6`

Output: `10`, `16.5`

Subtraction - You can subtract Two numbers with a minus sign `-`

Ex: `10 - 5`, `9.6 - 6.9`

Output: `5`, `2.7`

Multiplication - You can multiply Two numbers with an asterik `*`

Ex: `5 * 5`, `5.5 * 3`

Output: `25`, `16.5`

Division - You can divide Two numbers with a forward slash `/`

Ex: `50 / 5`, `20.4 / 4`

Output: `10`, `5.1`

Power - You can get a power of a number with 2 asterisks `**`

Ex: `2 ** 3`

Output: `8`

Modulo - You can get the remainder between Two numbers with a Percentage sign `%`

Ex: `69 % 6`, `25 % 5`

Output: `3`, `0`

Floor Divison: Ignores decimals when doing Division with two forward slashes `//`

Ex: `10 // 3`, `17 // 4`

Output: `3`, `4`

You can assign variables to use them for math

Ex:

---

```
1 a = 10
2 b = 5
3 print(a + b)
4 print(a - b)
5 print(a * b)
6 print(a / b)
7 print(a % b)
```

Output:

```
1 15
2 5
3 50
4 2
5 0
```

## Assingment Operators

Assignment operators are used to assign variables using operations. Here is a list of them.

```
+=
-
*=
/=
%=
//=
**=
```

Now let's have a few examples of using them

```
1 a = 15
2 b = 10
3
4 #After I print each value out pretend that I am resetting the value of a
5 a += b
6 print(a)
7 a -= b
8 print(a)
```

```
9 a *= b
10 print(a)
11 a /= b
12 print(a)
13 a %= b
14 print(a)
15 a //= b
16 print(a)
17 a **= b
18 print(a)
19 #I won't show the output for this because it is too big, but it would output the
```

Output:

```
1 25
2 5
3 150
4 1.5
5 5
6 1
7 Output of 15**10
```

The Assignment operators are technically just shortened down Basic Operators. For example `a += b` is just saying `a = a + b`, it's just another way to write it, however I still recommend using the shortened version.

## Comparison Operators

Comparison operators are used to get a true or false value. Here is a list of them

Checking if a value is equal to another value. If it is equal to the other value then it is True. Otherwise if the other value is not equal to the value it is False - `==`

Checking if a value is not equal to another value. If it is equal to the other value be compared then it is false. Otherwise if the value is not equal to the other value then it is False - `!=`

Checking if a value is greater than another value. If it is greater than the other value then it is True. Otherwise if the value is less than or equal to the other value then it is False - `>`

Checking if a value is less than another value. If it is less than the other value then it is true. Otherwise if the value is greater than or equal to the other value it is False - <

Checking if a value is greater than or equal to another value. If it is greater than or equal to the other value then it is True. Otherwise if the value is less than the other value then it is False -

>=

Checking if a value is less than or equal to another value. If it is less than or equal to the other value then it is True. Otherwise if the value is greater than the other value then it is False. - <=

Something to remember is that there are opposite comparison operators like == and != are opposite value because == is checking if the values are same and != is checking if the values are different.

Opposite Pairs:

== and !=  
> and <=  
< and >=

---

## Getting Inputs

You can also get the user to input something using the `input()` function, here's an example of using it:

```
input("How old are you: ")
```

Output

```
How old are you: 69
```

It would output `How old are you` and then I could say whatever my age was.

## Storing inputs in a variable

You can store the input that you get from a user into a variable like so

```
age = input("How old are you?")
```

Output:

```
How old are you: 69
```

Remember it doesn't print 69, I am just using 69 as an example input.

## Specifying input type

You can also specify the data type of the input like so

```
age = int(input("How old are you?"))
```

Output:

```
How old are you: 69
```

---

## Practice Problem 1

Small Exercise

Try to make an multiplication calculator where you ask the user to input 2 numbers and then output the product of the 2 numbers

Solution:

```
1 #Getting a number from the user
2 number1 = int(input("Enter a number"))
3 #Getting another from the user
4 number2 = int(input("Enter another number"))
5 #Adding the 2 numbers together
6 product = number1 * number2
7 #Printing out a message to the user telling them the product of the two numbers
8 print(f"The product of {number1} and {number2} is {product}")
```

## Lists

Lists are one of the more complex python data type. Lists hold other data types like strings, integers, floats, and even lists(We will go over nested lists in the next tutorial). They are declared using this syntax

```
fruits = ['apple','bannana','orange']
```

Each list item has an index, like strings. The index starts from 0.

Here's an example to show

```
1 fruits = ['apple','bannana','orange']
2 # 0 1 2
```

Lists also use negative indexing

```
1 fruits = ['apple','bannana','orange']
2 # -3 -2 -1
```

## List Methods

There are many methods that you can use with lists.

The `.append()` method adds an item to the list

Example:

```
1 fruits = ['apple','banana','orange']
2 fruits.append('mango')
```

Now fruits is `['apple','banana','orange','mango']`

The `.pop()` method removes an item from a specific index

Example:

```
1 fruits = ['apple','banana','orange','mango']
2 fruits.pop(3)
```

Now fruits is `['apple','banana','orange']`

The `.remove()` method remove a certain item from a list

Example:

```
1 fruits = ['apple','banana','orange','mango']
2 fruits.remove('orange')
```

Now fruits is `['apple','banana','mango']`

The `del` method can delete a whole list or a specific index of a list

Example:

```
1 fruits = ['apple','bannana','orange','mango']
2 del fruits[0]
```

Now fruits is ['bannana', 'orange', 'mango']

Another Example:

```
1 fruits = ['apple','bannana','orange','mango']
2 del fruits
```

Now there is no fruits list.

---

## Conditionals

Conditionals can be used to create code that will run if a certain condition will be used, the syntax is like so

```
1 if [condition]:
2 code
```

If you don't specify if you want the code to run if the condition is true or false then the default is that when your code is run, if the condition is true then the code will run.

Here's an example of a valid if statement using comparison operators

```
1 age = 69
2 if age == 69:
3 print("You are 69 years old")
```

Output:

```
You are 69 years old
```

## Another Example

```
1 age = 69
2 if age == 13:
3 print("You are 13 years old")
```

Output:

```
#Nothing because you are the variable age is not 13, and only if the age is 13,
```

## Elif

Elif is to have multiple conditions that can check if something is true if the if statement is not true. You can have as many elif statements as you want.

Example:

```
1 age = 9
2 if age == 13:
3 print("You are 13 years old")
4 elif age == 9:
5 print("You are 9 years old")
```

Output

```
You are 9 years old
```

## Another Example

```
1 age = 11
2 if age == 13:
3 print("You are 13 years old")
4 elif age == 4:
5 print("You are 4 years old")
```

Output:

```
#Nothing because in the first statement, age is not 13, so it won't run and in
```

## Else

Else conditions, are run when all of your if or elif statements are run, and haven't been executed.

Note: In if,elif, and else statements, in a single conditional, only one condition will run. So let's say you have multiple conditions, and 2 of them are true. The one that was stated the first will be run, and the second one won't be run.

Here's an example

```
1 age = 8
2 if age == 13:
3 print("You are 13 years old")
4 elif age == 9:
5 print("You are 9 years old")
6 else:
7 print("You are not 9 or 13 years old")
```

Another Example:

```
1 age = 13
2 name = "Bob"
3 if age == 13:
4 print("You are 13 years old")
5 elif age == 9:
6 print("You are 9 years old")
```

```
7 elif name == "Bob":
8 print("Your name is Bob")
9 elif name == "Mike":
10 print("Your name is Mike")
11 else:
12 print("You are not Mike or Bob, and you are not 13 or 9.")
```

Output:

```
You are 13 years old
```

Remember the reason it won't output "Your name is Bob" is because only one statement will run in a single conditional statement.

## pass

The `pass` keyword is used as a placeholder so when you have a if statement that you want nothing to execute if that condition is true then use the `pass` keyword

Example:

```
1 age = 15
2 if age == 15:
3 pass
4 else:
5 print("You are not 15")
```

This wouldn't output anything because nothing happens in the if statement because there is a `pass`

## Logic

You can have multiple conditions in each if statement, using the `and` keyword and the `or` keyword.

When you used the `and` keyword then the code will only run when all conditions are true.

When you use the `or` keyword then the code will run if one or more of the conditions are true.

---

## Practice Problem 2

### Small Exercise

Try to make an odd and even checker, where you get a number from user input and then the computer will check if the number is odd or even. If it is odd then print out the number and say that it is odd, and if it is even then print out the number and say that it is even. Go make a new repl and try it out! If it becomes too hard then feel free to check out the solution.

### Solution

```
1 #First Let's get the number input from the user, and let's call it number_input
2 number_input = int(input("What number Would you like to check?"))
3
4 #Then let's create the conditional Statements
5 #Checking if the number is even
6 if number_input % 2 == 0:
7 #Printing out the message that says the number is even
8 print(f"Your number {number_input} is even.")
9 #Now we'll have an else statement because if the number is not even then it must
10 else:
11 #Printing out the message that says the number is odd
12 print(f"Your number {number_input} is odd.")
```

---

## Loops

Loops are used to run things multiple times, and there are 2 main types, for loops and while loops

### For Loops

For loops are used to iterate through items in a list, dictionary, integer, or string. They can be used to do something for every item in a variable.

Example:

```
1 example_list = ['item1','other item','last item']
2 #we now have a list
3
4 for i in example_list: #loops through everything in the list
5 #and assigns the current thing to the variable i
6 print(i)
```

output:

```
1 item1
2 other item
3 last item
```

As you can see, it looped through every value in the list `example_list`.

The reason the `i` is there is to have a variable you can assign to the thing it is looping through. You can change `i` to any other name as long as another variable doesn't have the same name.

Another thing you can use is `range`.

```
1 for i in range(10):
2 print(i)
```

output:

```
1 0
2 1
3 2
4 3
5 4
```

```
6 5
7 6
8 7
9 8
10 9
```

It starts from 0 and goes to the number before the number you said, so it has 10 numbers in it.

If you want to exit a `for` loop, you can use `break`.

```
1 for i in range(10):
2 if i == 5:
3 break
4 else:
5 print(i)
```

output:

```
1 0
2 1
3 2
4 3
5 4
```

As you can see, when the number reached `5`, the loop ended.

You can also skip iterations using `continue`.

```
1 for i in range(10):
2 if i == 5:
3 continue
4 else:
5 print(i)
```

output:

```
1 0
2 1
3 2
4 3
5 4
6 6
7 7
8 8
9 9
```

When the loop reached 5, it skipped the print statement and went back to the beginning of the loop.

## While Loops

While loops are used to run a program while the variable stated is the condition stated.

Example:

```
1 num = 6
2 other = 1
3 while num > other:
4 print(other)
5 other += 1
```

output:

```
1 1
2 2
3 3
4 4
5 5
```

So, the loop keeps going on so long as `num` is greater than `other`, it prints `other`. But, when they were equal (both at 6), the loop stopped.

You can also use `break` and `continue` in while loops. It works the same as it did in the `for` loops.

The most common usage for `while` loops is with `bool` values.

```
1 var = True
2 while var:
3 print('hello')
```

**output:**

```
1 hello
2 hello
3 hello
4 hello
5 hello
6 hello
7 hello
8 hello
9 ...
```

Since `var` is never changed (it always stays true), the loop goes on forever.

---

## Practice Problem 3

Make a program where for every number from 1-100, if the number is divisible by 15, it would print Fizzbuzz, if the number is divisible by 5, it would print Fizz, if the number is divisible by 3 then bring Buzz, otherwise it will just print the number. Then Ask the user if they want to start the program again, if yes then restart it again, otherwise break out of the loop. Go make a new repl and try it out! If it becomes too hard then feel free to check the solutions.

Challenge: Ask the user the maximum number they want the program to run to.

**Solution:**

```
1 #A while loop to keep on running the program if the user wants it to
2 while True:
3 #A for loop to iterate through each number in the range of 1-100
```

```

4 for number in range(1,101):
5 #An if statement to check if the number is divisible by 15
6 if number % 15 == 0:
7 #Printing out Fizzbuzz
8 print("Fizzbuzz")
9 #An elif statement to check if the number is divisible by 5
10 elif number % 5 == 0:
11 #Printing out Fizz
12 print("Fizz")
13 #An elif statement to check if the number is divisible by 3
14 elif number % 3 == 0:
15 #Printing out Buzz
16 print("Buzz")
17 #An else statement to print out the number
18 else:
19 #Printing out the number
20 print(number)
21 #Asking the user if they want the program to run again
22 again = str(input("Would you like to run the program again? Enter Y/N: "))
23 #An if statement checking if the user said they want the program to run
24 if again.upper() == 'Y':
25 #Using pass as a placeholder because we want the program to continue
26 pass
27 #An else statement to break the while loop
28 else:
29 #Breaking the loop
30 break

```

Challenge solution:

```

1 #A while loop to keep on running the program if the user wants it to
2 while True:
3 #Asking the user what they would like the maximum number to be
4 max_range = input("What would you like the range to be? ")
5 #A for loop to iterate through each number in the range of 1- whatever number
6 for number in range(max_range):
7 #An if statement to check if the number is divisible by 15
8 if number % 15 == 0:
9 #Printing out Fizzbuzz
10 print("Fizzbuzz")
11 #An elif statement to check if the number is divisible by 5
12 elif number % 5 == 0:
13 #Printing out Fizz
14 print("Fizz")
15 #An elif statement to check if the number is divisible by 3
16 elif number % 3 == 0:
17 #Printing out Buzz
18 print("Buzz")
19 #An else statement to print out the number

```

```
20 else:
21 #Printing out the number
22 print(number)
23 #Asking the user if they want the program to run again
24 again = str(input("Would you like to run the program again? Enter Y/N: "))
25 #An if statement checking if the user said they want the program to run
26 if again.upper() == 'Y':
27 #Using pass as a placeholder because we want the program to continue
28 pass
29 #An else statement to break the while loop
30 else:
31 #Breaking the loop
32 break
```

## Escape Codes

There are multiple escape codes that you can use in your code to do different things

\n This prints the text on a new line.

Ex:

Code:

```
print("Hello\nTesting\n\nBackslash n")
```

Output:

```
1 Hello
2 Testing
3
4 Backslash n
```

\t This prints a tab.

Ex:

Code:

```
print('\t\ttab\t\tmore tabs')
```

Output:

```
tab more tabs
```

## Functions

A function is a block of code that runs code. They are mainly used when you want to use code again and again, without having to write it multiple times.

Here is the syntax for it

```
1 def function_name():
2 code_to_run
```

Here is an example:

```
1 def add():
2 sum = 5 + 5
3 print(sum)
```

The way that you use or call functions is through this syntax

```
add()
```

Output: 10

And you can use the function multiple times.

## return

There is a special keyword that can be used in functions and it is the `return` keyword

What the `return` keyword does, is that it gives the function that you call a value. And you have to print the function out when you want the code that was run

So here's an example

```
1 def try_func():
2 x = 5
3 return x
4 try_func()
```

You might think that using `try_func()` would print out 5, but really it would assign the value of 5 to the function. If you wanted it to print out 5, then you would have to do

```
print(tryfunc())
```

## Parameters

Parameters are something used in a function to make it more versatile and interactive.

Parameters are variables, that you pass in when trying to call a function.

Here's the syntax:

```
1 def function_name(parameters_needed):
2 code_to_run
```

Syntax to run it:

```
function_name(parameters)
```

Here's an example of a function using parameters

```
1 def add(num1, num2):
2 return int(num1) + int(num2)
3
4 print(add(9,10)) # 9 and 10 are parameters
```

this should output

```
19
```

Basically, the function is just saying, give me two numbers, and I'll add them and return the sum.

## Lambda

Lambdas are short functions that you can use in your code, and are assigned to variable names.

Here's the syntax

```
variable_name = lambda parameters : code_to_run
```

Calling syntax:

```
variable_name(parameters)
```

And here's an example

```
multiply_by5 = lambda num: num * 5
```

Calling it:

```
print(multiply_by5(5))
```

Output: 25

You can also use lambdas in function, they are mostly used in the return

```
1 def concatenator(string):
2 return lambda string2 : string2 + string
3
4 er_concatenator = concatenator("er")
5 print(er_concatenator("Program"))
```

Output: Programer

## Scopes

A scope is where a variable is stored. Like if it's inside of a function.

### Global

The global scope are variables that can be accessed by all of your code.

eg.

```
x = 'This is in the global scope'
```

## Local

The local scope is the scope of a function, and you may notice, that when you set a variable inside of a function then you can't access it outside the function, and you will get an error. This happens because the version of that variable only changes inside of the function. Here is an example of the error.

```
1 def y():
2 x = 1
3
4 y()
5 print(x)
```

The way you can fix this is by using the global keyword and declaring that variable as a global variable

```
1 def y():
2 global x
3 # Declaring x to be a global variable
4 x += 1
5
6 y()
7 print(x)
```

The `global` keyword basically moves the `x` variable from the local scope to the global scope, and all changes that are made when the function is called is applied to the global version of `x`.

**Wk-17**

# Untitled



CS47 Strings and Arrays Python IV w/ Tom Tarpey

<https://youtu.be/AKDIKZ6zwmw>



Strings and Arrays.ipynb

<https://gist.github.com/bgoonz/518b1ceb17c47ca236c1522445ad417c#file-strings-and-arrays-ipynb>

Keywords:

```
1 ***and del for is raise
2 assert elif from lambda return
3 break else global not try
4 class except if or while
5 continue exec import pass
6 def finally in print***
7
```

py-notes.pdf

<https://bryan-guner.gitbook.io/notesarchive/>

---

DOCS:

<https://docs.python.org/3/>

```
1 import math
2
3 def say_hi(name):
4 """<---- Multi-Line Comments and Docstrings
5 This is where you put your content for help() to inform the user
```

```
6 about what your function does and how to use it
7 """
8 print(f"Hello {name}!")
9
10 print(say_hi("Bryan")) # Should get the print inside the function, then None
11 # Boolean Values
12 # Work the same as in JS, except they are title case: True and False
13 a = True
14 b = False
15 # Logical Operators
16 # != not, || = or, && = and
17 print(True and True)
18 print(True and not True)
19 print(True or True)
20 # Truthiness - Everything is True except...
21 # False - None, False, '', [], (), set(), range(0)
22 # Number Values
23 # Integers are numbers without a floating decimal point
24 print(type(3)) # type returns the type of whatever argument you pass in
25 # Floating Point values are numbers with a floating decimal point
26 print(type(3.5))
27 # Type Casting
28 # You can convert between ints and floats (along with other types...)
29 print(float(3)) # If you convert a float to an int, it will truncate the decimal
30 print(int(4.5))
31 print(type(str(3)))
32 # Python does not automatically convert types like JS
33 # print(17.0 + ' heyooo ' + 17) # TypeError
34 # Arithmetic Operators
35 # ** - exponent (comparable to Math.pow(num, pow))
36 # // - integer division
37 # There is no ++ or -- in Python
38 # String Values
39 # We can use single quotes, double quotes, or f' ' for string formats
40 # We can use triple single quotes for multiline strings
41 print(
42 """This here's a story
43 All about how
44 My life got twist
45 Turned upside down
46 """
47)
48 # Three double quotes can also be used, but we typically reserve these for
49 # multi-line comments and function docstrings (refer to lines 6-9)(Nice :D)
50 # We use len() to get the length of something
51 print(len("Bryan G")) # 7 characters
52 print(len(["hey", "ho", "hey", "hey", "ho"])) # 5 list items
53 print(len({1, 2, 3, 4, 5, 6, 7, 9})) # 8 set items
54 # We can index into strings, list, etc..self.
55 name = "Bryan"
56 for i in range(len(name)):
57 print(name[i]) # B, r, y, a, n
58 # We can index starting from the end as well, with negatives
```

```
59 occupation = "Full Stack Software Engineer"
60 print(occupation[-3]) # e
61 # We can also get ranges in the index with the [start:stop:step] syntax
62 print(occupation[0:4:1]) # step and stop are optional, stop is exclusive
63 print(occupation[::-4]) # beginning to end, every 4th letter
64 print(occupation[4:14:2]) # Let's get weird with it!
65 # NOTE: Indexing out of range will give you an IndexError
66 # We can also get the index of things with the .index() method, similar to indexOf
67 print(occupation.index("Stack"))
68 print(["Mike", "Barry", "Cole", "James", "Mark"].index("Cole"))
69 # We can count how many times a substring/item appears in something as well
70 print(occupation.count("S"))
71 print(
72 """Now this here's a story all about how
73 My life got twist turned upside down
74 I forget the rest but the the potato
75 smells like the potato""".count(
76 "the"
77)
78)
79 # We concatenate the same as Javascript, but we can also multiply strings
80 print("dog " + "show")
81 print("ha" * 10)
82 # We can use format for a multitude of things, from spaces to decimal places
83 first_name = "Bryan"
84 last_name = "Guner"
85 print("Your name is {} {}".format(first_name, last_name))
86 # Useful String Methods
87 print("Hello".upper()) # HELLO
88 print("Hello".lower()) # hello
89 print("HELLO".islower()) # False
90 print("HELLO".isupper()) # True
91 print("Hello".startswith("he")) # False
92 print("Hello".endswith("lo")) # True
93 print("Hello There".split()) # [Hello, There]
94 print("hello1".isalpha()) # False, must consist only of letters
95 print("hello1".isalnum()) # True, must consist of only letters and numbers
96 print("3215235123".isdecimal()) # True, must be all numbers
97 # True, must consist of only spaces/tabs/newlines
98 print("\n".isspace())
99 # False, index 0 must be upper case and the rest lower
100 print("Bryan Guner".istitle())
101 print("Michael Lee".istitle()) # True!
102 # Duck Typing - If it walks like a duck, and talks like a duck, it must be a duck
103 # Assignment - All like JS, but there are no special keywords like let or const
104 a = 3
105 b = a
106 c = "heyoo"
107 b = ["reassignment", "is", "fine", "G!"]
108 # Comparison Operators - Python uses the same equality operators as JS, but no ==
109 # < - Less than
110 # > - Greater than
111 # <= - Less than or Equal
```

```
112 # >= - Greater than or Equal
113 # == - Equal to
114 # != - Not equal to
115 # is - Refers to exact same memory location
116 # not - !
117 # Precedence - Negative Signs(not) are applied first(part of each number)
118 # - Multiplication and Division(and) happen next
119 # - Addition and Subtraction(or) are the last step
120 # NOTE: Be careful when using not along with ==
121 print(not a == b) # True
122 # print(a == not b) # Syntax Error
123 print(a == (not b)) # This fixes it. Answer: False
124 # Python does short-circuit evaluation
125 # Assignment Operators - Mostly the same as JS except Python has **= and //=
126 # Flow Control Statements - if, while, for
127 # Note: Python smushes 'else if' into 'elif'!
128 if 10 < 1:
129 print("We don't get here")
130 elif 10 < 5:
131 print("Nor here...")
132 else:
133 print("Hey there!")
134 # Looping over a string
135 for c in "abcdefghijklm":
136 print(c)
137 # Looping over a range
138 for i in range(5):
139 print(i + 1)
140 # Looping over a list
141 lst = [1, 2, 3, 4]
142 for i in lst:
143 print(i)
144 # Looping over a dictionary
145 spam = {"color": "red", "age": 42, "items": [(1, "hey"), (2, "hooo!")]}
146 for v in spam.values():
147 print(v)
148 # Loop over a list of tuples and destructuring the values
149 # Assuming spam.items returns a list of tuples each containing two items (k, v)
150 for k, v in spam.items():
151 print(f"{k}: {v}")
152 # While loops as long as the condition is True
153 # - Exit loop early with break
154 # - Exit iteration early with continue
155 spam = 0
156 while True:
157 print("Sike That's the wrong Numba")
158 spam += 1
159 if spam < 5:
160 continue
161 break
162
163 # Functions - use def keyword to define a function in Python
164
```

```
165 def printCopyright():
166 print("Copyright 2021, Bgoonz")
167
168 # Lambdas are one liners! (Should be at least, you can use parenthesis to dis-
169 def avg(num1, num2):
170 return print(num1 + num2)
171
172 avg(1, 2)
173 # Calling it with keyword arguments, order does not matter
174 avg(num2=20, num1=1252)
175 printCopyright()
176 # We can give parameters default arguments like JS
177
178 def greeting(name, saying="Hello"):
179 print(saying, name)
180
181 greeting("Mike") # Hello Mike
182 greeting("Bryan", saying="Hello there...")
183 # A common gotcha is using a mutable object for a default parameter
184 # All invocations of the function reference the same mutable object
185
186 def append_item(item_name, item_list=[]): # Will it obey and give us a new l-
187 item_list.append(item_name)
188 return item_list
189
190 # Uses same item list unless otherwise stated which is counterintuitive
191 print(append_item("notebook"))
192 print(append_item("notebook"))
193 print(append_item("notebook", []))
194 # Errors - Unlike JS, if we pass the incorrect amount of arguments to a funct-
195 # it will throw an error
196 # avg(1) # TypeError
197 # avg(1, 2, 2) # TypeError
198 # ----- DAY 2 -----
199 # Functions - * to get rest of position arguments as tuple
200 # - ** to get rest of keyword arguments as a dictionary
201 # Variable Length positional arguments
202
203 def add(a, b, *args):
204 # args is a tuple of the rest of the arguments
205 total = a + b
206 for n in args:
207 total += n
208 return total
209
210 print(add(1, 2)) # args is None, returns 3
211 print(add(1, 2, 3, 4, 5, 6)) # args is (3, 4, 5, 6), returns 21
212 # Variable Length Keyword Arguments
213
214 def print_names_and_countries(greeting, **kwargs):
215 # kwargs is a dictionary of the rest of the keyword arguments
216 for k, v in kwargs.items():
217 print(greeting, k, "from", v)
```

```
218
219 print_names_and_countries(
220 "Hey there", Monica="Sweden", Mike="The United States", Mark="China"
221)
222 # We can combine all of these together
223
224 def example2(arg1, arg2, *args, kw_1="cheese", kw_2="horse", **kwargs):
225 pass
226
227 # Lists are mutable arrays
228 empty_list = []
229 roomates = ["Beau", "Delynn"]
230 # List built-in function makes a list too
231 specials = list()
232 # We can use 'in' to test if something is in the list, like 'includes' in JS
233 print(1 in [1, 2, 4]) # True
234 print(2 in [1, 3, 5]) # False
235 # Dictionaries - Similar to JS POJO's or Map, containing key value pairs
236 a = {"one": 1, "two": 2, "three": 3}
237 b = dict(one=1, two=2, three=3)
238 # Can use 'in' on dictionaries too (for keys)
239 print("one" in a) # True
240 print(3 in b) # False
241 # Sets - Just like JS, unordered collection of distinct objects
242 bedroom = {"bed", "tv", "computer", "clothes", "playstation 4"}
243 # bedroom = set("bed", "tv", "computer", "clothes", "playstation 5")
244 school_bag = set(
245 ["book", "paper", "pencil", "pencil", "book", "book", "book", "eraser"]
246)
247 print(school_bag)
248 print(bedroom)
249 # We can use 'in' on sets as well
250 print(1 in {1, 2, 3}) # True
251 print(4 in {1, 3, 5}) # False
252 # Tuples are immutable lists of items
253 time_blocks = ("AM", "PM")
254 colors = "red", "green", "blue" # Parenthesis not needed but encouraged
255 # The tuple built-in function can be used to convert things to tuples
256 print(tuple("abc"))
257 print(tuple([1, 2, 3]))
258 # 'in' may be used on tuples as well
259 print(1 in (1, 2, 3)) # True
260 print(5 in (1, 4, 3)) # False
261 # Ranges are immutable lists of numbers, often used with for loops
262 # - start - default: 0, first number in sequence
263 # - stop - required, next number past last number in sequence
264 # - step - default: 1, difference between each number in sequence
265 range1 = range(5) # [0,1,2,3,4]
266 range2 = range(1, 5) # [1,2,3,4]
267 range3 = range(0, 25, 5) # [0,5,10,15,20]
268 range4 = range(0) # []
269 for i in range1:
270 print(i)
```

```

271 # Built-in functions:
272 # Filter
273
274 def isOdd(num):
275 return num % 2 == 1
276
277 filtered = filter(isOdd, [1, 2, 3, 4])
278 print(list(filtered))
279 for num in filtered:
280 print(f"first way: {num}")
281 print("--" * 20)
282 [print(f"list comprehension: {i}") for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 1]
283
284 # Map
285
286 def toUpper(str):
287 return str.upper()
288
289 upperCased = map(toUpper, ["a", "b", "c", "d"])
290 print(list(upperCased))
291
292 sorted_items = sorted(["john", "tom", "sonny", "Mike"])
293 print(list(sorted_items)) # Notice uppercase comes before lowercase
294
295 # Using a key function to control the sorting and make it case insensitive
296 sorted_items = sorted(["john", "tom", "sonny", "Mike"], key=str.lower)
297 print(sorted_items)
298
299 # You can also reverse the sort
300 sorted_items = sorted(["john", "tom", "sonny", "Mike"],
301 key=str.lower, reverse=True)
302 print(sorted_items)
303
304 # Enumerate creates a tuple with an index for what you're enumerating
305 quarters = ["First", "Second", "Third", "Fourth"]
306 print(list(enumerate(quarters)))
307 print(list(enumerate(quarters, start=1)))
308
309 # Zip takes list and combines them as key value pairs, or really however you want
310 keys = ("Name", "Email")
311 values = ("Buster", "cheetoh@johhnydepp.com")
312 zipped = zip(keys, values)
313 print(list(zipped))
314
315 # You can zip more than 2
316 x_coords = [0, 1, 2, 3, 4]
317 y_coords = [4, 6, 10, 9, 10]
318 z_coords = [20, 10, 5, 9, 1]
319 coords = zip(x_coords, y_coords, z_coords)
320 print(list(coords))
321
322 # Len reports the length of strings along with list and any other object data
323 # doing this to save myself some typing
324
325 def print_len(item):
326 return print(len(item))
327
328 print_len("Mike")
329 print_len([1, 5, 2, 10, 3, 10])

```

```
324 print_len({1, 5, 10, 9, 10}) # 4 because there is a duplicate here (10)
325 print_len((1, 4, 10, 9, 20))
326 # Max will return the max number in a given scenario
327 print(max(1, 2, 35, 1012, 1))
328 # Min
329 print(min(1, 5, 2, 10))
330 print(min([1, 4, 7, 10]))
331 # Sum
332 print(sum([1, 2, 4]))
333 # Any
334 print(any([True, False, False]))
335 print(any([False, False, False]))
336 # All
337 print(all([True, True, False]))
338 print(all([True, True, True]))
339 # Dir returns all the attributes of an object including it's methods and dunder
340 user = {"Name": "Bob", "Email": "bob@bob.com"}
341 print(dir(user))
342 # Importing packages and modules
343 # - Module - A Python code in a file or directory
344 # - Package - A module which is a directory containing an __init__.py file
345 # - Submodule - A module which is contained within a package
346 # - Name - An exported function, class, or variable in a module
347 # Unlike JS, modules export ALL names contained within them without any special
348 # Assuming we have the following package with four submodules
349 # math
350 # | __init__.py
351 # | addition.py
352 # | subtraction.py
353 # | multiplication.py
354 # | division.py
355 # If we peek into the addition.py file we see there's an add function
356 # addition.py
357 # We can import 'add' from other places because it's a 'name' and is automatically
358
359 # def add(num1, num2):
360 # return num1 + num2
361
362 # Notice the . syntax because this package can import it's own submodules.
363 # Our __init__.py has the following files
364 # This imports the 'add' function
365 # And now it's also re-exported in here as well
366 # from .addition import add
367 # These import and re-export the rest of the functions from the submodule
368 # from .subtraction import subtract
369 # from .division import divide
370 # from .multiplication import multiply
371 # So if we have a script.py and want to import add, we could do it many ways
372 # This will load and execute the 'math/__init__.py' file and give
373 # us an object with the exported names in 'math/__init__.py'
374 # print(math.add(1,2))
375 # This imports JUST the add from 'math/__init__.py'
376 # from math import add
```

```
377 # print(add(1, 2))
378 # This skips importing from 'math/__init__.py' (although it still runs)
379 # and imports directly from the addition.py file
380 # from math.addition import add
381 # This imports all the functions individually from 'math/__init__.py'
382 # from math import add, subtract, multiply, divide
383 # print(add(1, 2))
384 # print(subtract(2, 1))
385 # This imports 'add' renames it to 'add_some_numbers'
386 # from math import add as add_some_numbers
387 # ----- DAY 3 -----
388 # Classes, Methods, and Properties
389
390 class AngryBird:
391 # Slots optimize property access and memory usage and prevent you
392 # from arbitrarily assigning new properties to the instance
393 __slots__ = ["_x", "_y"]
394 # Constructor
395
396 def __init__(self, x=0, y=0):
397 # Doc String
398 """
399 Construct a new AngryBird by setting its position to (0, 0)
400 """
401 # Instance Variables
402 self._x = x
403 self._y = y
404
405 # Instance Method
406
407 def move_up_by(self, delta):
408 self._y += delta
409
410 # Getter
411
412 @property
413 def x(self):
414 return self._x
415
416 # Setter
417
418 @x.setter
419 def x(self, value):
420 if value < 0:
421 value = 0
422 self._x = value
423
424 @property
425 def y(self):
426 return self._y
427
428 @y.setter
429 def y(self, value):
```

```

430 self._y = value
431
432 # Dunder Repr... called by 'print'
433
434 def __repr__(self):
435 return f"<AngryBird ({self._x}, {self._y})>"
436
437 # JS to Python Classes cheat table
438 # JS Python
439 # constructor() def __init__(self):
440 # super() super().__init__()
441 # this.property self.property
442 # this.method self.method()
443 # method(arg1, arg2){} def method(self, arg1, ...)
444 # get someProperty(){} @property
445 # set someProperty(){} @someProperty.setter
446 # List Comprehensions are a way to transform a list from one format to another
447 # - Pythonic Alternative to using map or filter
448 # - Syntax of a list comprehension
449 # - new_list = [value loop condition]
450 # Using a for loop
451 squares = []
452 for i in range(10):
453 squares.append(i ** 2)
454 print(squares)
455 # value = i ** 2
456 # loop = for i in range(10)
457 squares = [i ** 2 for i in range(10)]
458 print(list(squares))
459 sentence = "the rocket came back from mars"
460 vowels = [character for character in sentence if character in "aeiou"]
461 print(vowels)
462 # You can also use them on dictionaries. We can use the items() method
463 # for the dictionary to loop through it getting the keys and values out at once
464 person = {"name": "Corina", "age": 32, "height": 1.4}
465 # This loops through and capitalizes the first letter of all keys
466 newPerson = {key.title(): value for key, value in person.items()}
467 print(list(newPerson.items()))

```

## 2.1.7 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the



(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer – the indentation of `return r` does not match a level popped off the stack.)

<https://ds-unit-5-lambda.netlify.app/>

## Python Study Guide for a JavaScript Programmer

Bryan Guner

Mar 5 · 15 min read

Main data types	List operations	List methods
<pre>boolean = True / False integer = 10 float = 10.01 string = "123abc" list = [value1, value2, ...] dictionary = {key1:value1, key2:value2, ...}</pre>	<pre>list = [] defines an empty list list[i] = x stores x with index i list[i] retrieves the item with index i list[-1] retrieves last item list[i:j] retrieves items in the range i to j del list[i] removes the item with index i</pre>	<pre>list.append(x) adds x to the end of the list list.extend(L) appends L to the end of the list list.insert(i,x) inserts x at i position list.remove(x) removes the first list item whose value is x list.pop(i) removes the item at position i and returns its value list.clear() removes all items from the list list.index(x) returns a list of values delimited by x list.count(x) returns a string with list values joined by S list.sort() sorts list items list.reverse() reverses list elements list.copy() returns a copy of the list</pre>
Numeric operators	Dictionary operations	String methods
<pre>+ addition - subtraction * multiplication / division ** exponent % modulus // floor division</pre>	<pre>dict = {} defines an empty dictionary dict[k] = x stores x associated to key k dict[k] retrieves the item with key k del dict[k] removes the item with key k</pre>	<pre>string.upper() converts to uppercase string.lower() converts to lowercase string.count(x) counts how many times x appears string.find(x) position of the x first occurrence string.replace(x,y) replaces x for y string.strip(x) returns a list of values delimited by x string.join(L) returns a string with L values joined by string string.format(x) returns a string that includes formatted x</pre>
Boolean operators	Special characters	Dictionary methods
<pre>and logical AND or logical OR not logical NOT</pre>	<pre># coment \n new line \&lt;char&gt; escape char</pre>	<pre>dict.keys() returns a list of keys dict.values() returns a list of values dict.items() returns a list of pairs (key,value) dict.get(k) returns the value associated to the key k dict.pop() removes the item associated to the key and returns its value dict.update(D) adds keys-values (D) to dictionary dict.clear() removes all keys-values from the dictionary dict.copy() returns a copy of the dictionary</pre>
String operations		
<pre>string[i] retrieves character at position i string[-1] retrieves last character string[i:j] retrieves characters in range i to j</pre>		

Legend: x,y stand for any kind of data values, s for a string, n for a number, L for a list where i,j are list indexes, D stands for a dictionary and k is a dictionary key.

[https://miro.medium.com/max/1400/1\\*3V9VOfPk\\_hrFdbEAd3j-QQ.png](https://miro.medium.com/max/1400/1*3V9VOfPk_hrFdbEAd3j-QQ.png)

## Applications of Tutorial & Cheat Sheet Respectively (At Bottom Of Tutorial):

## Basics

- **PEP8** : Python Enhancement Proposals, style-guide for Python.
- `print` is the equivalent of `console.log` .

```
'print() == console.log()'
```

### # is used to make comments in your code.

```
1 def foo():
2 """
3 The foo function does many amazing things that you
4 should not question. Just accept that it exists and
5 use it with caution.
6 """
7 secretThing()
```

Python has a built in help function that let's you see a description of the source code without having to navigate to it... “-SickNasty ... Autor Unknown”

## Numbers

- Python has three types of numbers:

1. Integer
2. Positive and Negative Counting Numbers.

No Decimal Point

Created by a literal non-decimal point number ... or ... with the `int()` constructor.

```
1 print(3) # => 3
2 print(int(19)) # => 19
```

```
3 print(int()) # => 0
```

### 3. Complex Numbers

Consist of a real part and imaginary part.

## Boolean is a subtype of integer in Python.□□

If you came from a background in JavaScript and learned to accept the premise(s) of the following meme...

Than I am sure you will find the means to suspend your disbelief.

```
1 print(2.24) # => 2.24
2 print(2.) # => 2.0
3 print(float()) # => 0.0
4 print(27e-5) # => 0.00027
```

## KEEP IN MIND:

The i is switched to a j in programming.

*This is because the letter i is common place as the de facto index for any and all enumerable entities so it just makes sense not to compete for name-space when there's another 25 letters that don't get used for every loop under the sun. My most medium apologies to Leonhard Euler.*

```
1 print(7j) # => 7j
2 print(5.1+7.7j)) # => 5.1+7.7j
3 print(complex(3, 5)) # => 3+5j
4 print(complex(17)) # => 17+0j
5 print(complex()) # => 0j
```

- **Type Casting** : The process of converting one number to another.

```
1 # Using Float
2 print(17) # => 17
3 print(float(17)) # => 17.0# Using Int
4 print(17.0) # => 17.0
5 print(int(17.0)) # => 17# Using Str
6 print(str(17.0) + ' and ' + str(17)) # => 17.0 and 17
```

The arithmetic operators are the same between JS and Python, with two additions:

- `**`: Double asterisk for exponent.
- `//`: Integer Division.
- There are no spaces between math operations in Python.
- Integer Division gives the other part of the number from Module; it is a way to do round down numbers replacing `Math.floor()` in JS.
- There are no `++` and `--` in Python, the only shorthand operators are:

---

## Strings

- Python uses both single and double quotes.
- You can escape strings like so `'Jodi asked, "What\\\'s up, Sam?"'`
- Multiline strings use triple quotes.

```
1 print('''My instructions are very long so to make them
2 more readable in the code I am putting them on
3 more than one line. I can even include "quotes"
4 of any kind because they won't get confused with
5 the end of the string!'''')
```

Use the `len()` function to get the length of a string.

```
print(len("Spaghetti")) # => 9
```

# Python uses zero-based indexing

## Python allows negative indexing (thank god!)

```
print("Spaghetti)[-1] # => i print("Spaghetti)[-4] # => e
```

- Python let's you use ranges

You can think of this as roughly equivalent to the slice method called on a JavaScript object or string... (*mind you that in JS ... strings are wrapped in an object (under the hood)... upon which the string methods are actually called. As a immutable privative type by textbook definition, a string literal could not hope to invoke most of its methods without violating the state it was bound to on initialization if it were not for this bit of syntactic sugar.*)

```
1 print("Spaghetti"[1:4]) # => pag
2 print("Spaghetti"[4:-1]) # => hett
3 print("Spaghetti"[4:4]) # => (empty string)
```

- The end range is exclusive just like `slice` in JS.

```
1 # Shortcut to get from the beginning of a string to a certain index.
2 print("Spaghetti)[:4] # => Spag
3 print("Spaghetti":-1) # => Spaghett# Shortcut to get from a certain index
4 print("Spaghetti")[1:]) # => paghetti
5 print("Spaghetti)[-4:] # => etti
```

- The `index` string function is the equiv. of `indexof()` in JS

```
1 print("Spaghetti".index("h")) # => 4
2 print("Spaghetti".index("t")) # => 6
```

- The `count` function finds out how many times a substring appears in a string... pretty nifty for a hard coded feature of the language.

```

1 print("Spaghetti".count("h")) # => 1
2 print("Spaghetti".count("t")) # => 2
3 print("Spaghetti".count("s")) # => 0
4 print('''We choose to go to the moon in this decade and do the other things,
5 not because they are easy, but because they are hard, because that goal will
6 serve to organize and measure the best of our energies and skills, because tha
7 challenge is one that we are willing to accept, one we are unwilling to
8 postpone, and one which we intend to win, and the others, too.
9 '''.count('the ')) # => 4

```

- You can use `+` to concatenate strings, just like in JS.
- You can also use `*` to repeat strings or multiply strings.
- Use the `format()` function to use placeholders in a string to input values later on.

```

1 first_name = "Billy"
2 last_name = "Bob"
3 print('Your name is {0} {1}'.format(first_name, last_name)) # => Your name is

```

- Shorthand way to use format function is:*

```
print(f'Your name is {first_name} {last_name}')
```

## Some useful string methods.

- Note that in JS `join` is used on an Array, in Python it is used on String.

Value	Method	Result
s = "Hello"	s.upper()	"HELLO"
s = "Hello"	s.lower()	"hello"
s = "Hello"	s.islower()	False
s = "hello"	s.islower()	True
s = "Hello"	s.isupper()	False
s = "HELLO"	s.isupper()	True
s = "Hello"	s.startswith("He")	True
s = "Hello"	s.endswith("lo")	True
s = "Hello World"	s.split()	["Hello", "World"]
s = "i-am-a-dog"	s.split("-")	["i", "am", "a", "dog"]

[https://miro.medium.com/max/630/0\\*eE3E5H0AoqkhqK1z.png](https://miro.medium.com/max/630/0*eE3E5H0AoqkhqK1z.png)

- There are also many handy testing methods.

Method	Purpose
isalpha()	returns True if the string consists only of letters and is not blank.
isalnum()	returns True if the string consists only of letters and numbers and is not blank.
isdecimal()	returns True if the string consists only of numeric characters and is not blank.
isspace()	returns True if the string consists only of spaces, tabs, and newlines and is not blank.
istitle()	returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

[https://miro.medium.com/max/630/0\\*Q0CMqFd4PozLDFPB.png](https://miro.medium.com/max/630/0*Q0CMqFd4PozLDFPB.png)

## Variables and Expressions

- **Duck-Typing** : Programming Style which avoids checking an object's type to figure out what it can do.
- Duck Typing is the fundamental approach of Python.
- Assignment of a value automatically declares a variable.

```
1 a = 7
```

```
2 b = 'Marbles'
3 print(a) # => 7
4 print(b) # => Marbles
```

- You can chain variable assignments to give multiple var names the same value.

## Use with caution as this is highly unreadable

```
1 count = max = min = 0
2 print(count) # => 0
3 print(max) # => 0
4 print(min) # => 0
```

## The value and type of a variable can be re-assigned at any time.

```
1 a = 17
2 print(a) # => 17
3 a = 'seventeen'
4 print(a) # => seventeen
```

- *NaN* does not exist in Python, but you can 'create' it like so: `print(float("nan"))`
- Python replaces `null` with `None`.
- `None` is an object and can be directly assigned to a variable.

Using `None` is a convenient way to check to see why an action may not be operating correctly in your program.

## Boolean Data Type

- One of the biggest benefits of Python is that it reads more like English than JS does.

Python	JavaScript
and	&&
or	
not	!

[https://miro.medium.com/max/1400/0\\*HQpndNhm1Z\\_xSoHb.png](https://miro.medium.com/max/1400/0*HQpndNhm1Z_xSoHb.png)

```

1 # Logical AND
2 print(True and True) # => True
3 print(True and False) # => False
4 print(False and False) # => False# Logical OR
5 print(True or True) # => True
6 print(True or False) # => True
7 print(False or False) # => False# Logical NOT
8 print(not True) # => False
9 print(not False and True) # => True
10 print(not True or False) # => False

```

- By default, Python considers an object to be true UNLESS it is one of the following:
- Constant `None` or `False`
- Zero of any numeric type.
- Empty Sequence or Collection.
- `True` and `False` must be capitalized

## Comparison Operators

- Python uses all the same equality operators as JS.
- In Python, equality operators are processed from left to right.
- Logical operators are processed in this order:

1. **NOT**
2. **AND**
3. **OR**

Just like in JS, you can use parentheses to change the inherent order of operations. Short Circuit : Stopping a program when a true or false has been reached.

Expression	Right side evaluated?
<code>True</code> and ...	Yes
<code>False</code> and ...	No
<code>True</code> or ...	No
<code>False</code> or ...	Yes

[https://miro.medium.com/max/630/0\\*qHzGRLTOMTf30miT.png](https://miro.medium.com/max/630/0*qHzGRLTOMTf30miT.png)

---

## Identity vs Equality

```

1 print (2 == '2') # => False
2 print (2 is '2') # => Falseprint ("2" == '2') # => True
3 print ("2" is '2') # => True# There is a distinction between the number type
4 print (2 == 2.0) # => True
5 print (2 is 2.0) # => False

```

- In the Python community it is better to use `is` and `is not` over `==` or `!=`

## If Statements

```

if name == 'Monica': print('Hi, Monica.')
if name == 'Monica': print('Hi, Monica.')
else: print('Hello, stranger.')
if name == 'Monica': print('Hi, Monica.')
elif age < 12: print('You are not Monica, kiddo.')
elif age > 2000:
 print('Unlike you, Monica is not an undead, immortal vampire.')
elif age > 100:
 print('You are not Monica, grannie.')

```

*Remember the order of `elif` statements matter.*

---

## While Statements

```

1 spam = 0
2 while spam < 5:
3 print('Hello, world.')
4 spam = spam + 1

```

- `Break` statement also exists in Python.

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam >= 5:
6 break
```

- As are `continue` statements

```
1 spam = 0
2 while True:
3 print('Hello, world.')
4 spam = spam + 1
5 if spam < 5:
6 continue
7 break
```

## Try/Except Statements

- Python equivalent to `try/catch`

```
1 a = 321
2 try:
3 print(len(a))
4 except:
5 print('Silently handle error here') # Optionally include a correction
6 a = str(a)
7 print(len(a)a = '321'
8 try:
9 print(len(a))
10 except:
11 print('Silently handle error here') # Optionally include a correction
12 a = str(a)
13 print(len(a))
```

- You can name an error to give the output more specificity.

```
1 a = 100
2 b = 0
3 try:
4 c = a / b
5 except ZeroDivisionError:
6 c = None
7 print(c)
```

- You can also use the `pass` command to bypass a certain error.

```
1 a = 100
2 b = 0
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
```

- The `pass` method won't allow you to bypass every single error so you can chain an exception series like so:

```
1 a = 100
2 # b = "5"
3 try:
4 print(a / b)
5 except ZeroDivisionError:
6 pass
7 except (TypeError, NameError):
8 print("ERROR!")
```

- You can use an `else` statement to end a chain of `except` statements.

```
1 # tuple of file names
2 files = ('one.txt', 'two.txt', 'three.txt')# simple loop
```

```

3 for filename in files:
4 try:
5 # open the file in read mode
6 f = open(filename, 'r')
7 except OSError:
8 # handle the case where file does not exist or permission is denied
9 print('cannot open file', filename)
10 else:
11 # do stuff with the file object (f)
12 print(filename, 'opened successfully')
13 print('found', len(f.readlines()), 'lines')
14 f.close()

```

- `finally` is used at the end to clean up all actions under any circumstance.

```

1 def divide(x, y):
2 try:
3 result = x / y
4 except ZeroDivisionError:
5 print("Cannot divide by zero")
6 else:
7 print("Result is", result)
8 finally:
9 print("Finally...")

```

- Using duck typing to check to see if some value is able to use a certain method.

```

1 # Try a number - nothing will print out
2 a = 321
3 if hasattr(a, '__len__'):
4 print(len(a))# Try a string - the length will print out (4 in this case)
5 b = "5555"
6 if hasattr(b, '__len__'):
7 print(len(b))

```

## Pass

- Pass Keyword is required to write the JS equivalent of :

```
1 if (true) {
2 }while (true) {}if True:
3 passwhile True:
4 pass
```

## Functions

- Function definition includes:
- The `def` keyword
- The name of the function
- A list of parameters enclosed in parentheses.
- A colon at the end of the line.
- One tab indentation for the code to run.
- You can use default parameters just like in JS

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)greeting("Monica")
3 # Hello Monica
4 greeting("Barry", "Hey")
5 # Hey Barry
```

**Keep in mind, default parameters must always come after regular parameters.**

```
1 # THIS IS BAD CODE AND WILL NOT RUN
2 def increment(delta=1, value):
3 return delta + value
```

- You can specify arguments by name without destructuring in Python.

```
1 def greeting(name, saying="Hello"):
2 print(saying, name)# name has no default value, so just provide the value
3 # saying has a default value, so use a keyword argument
```

```
4 greeting("Monica", saying="Hi")
```

- The `lambda` keyword is used to create anonymous functions and are supposed to be `one-liners`.

```
toUpper = lambda s: s.upper()
```

## Notes

### Formatted Strings

Remember that in Python `join()` is called on a string with an array/list passed in as the argument. Python has a very powerful formatting engine. `format()` is also applied directly to strings.

```
1 shopping_list = ['bread', 'milk', 'eggs']
2 print(', '.join(shopping_list))
```

### Comma Thousands Separator

```
1 print('{:,}'.format(1234567890))
2 '1,234,567,890'
```

### Date and Time

```
1 d = datetime.datetime(2020, 7, 4, 12, 15, 58)
2 print('{:%Y-%m-%d %H:%M:%S}'.format(d))
```

## Percentage

```
1 points = 190
2 total = 220
3 print('Correct answers: {:.2%}'.format(points/total))
4 Correct answers: 86.36%
```

## Data Tables

```
1 width=8
2 print(' decimal hex binary')
3 print('-'*27)
4 for num in range(1,16):
5 for base in 'dXb':
6 print('{0:{width}}{base}'.format(num, base=base, width=width), end=' ')
7 print()
8 Getting Input from the Command Line
9 Python runs synchronously, all programs and processes will stop when listening
10 The input function shows a prompt to a user and waits for them to type 'ENTER'
11 Scripts vs Programs
12 Programming Script : A set of code that runs in a linear fashion.
13 The largest difference between scripts and programs is the level of complexity
```

\*\*Python can be used to display html, css, and JS.\*\**It is common to use Python as an API (Application Programming Interface)*

## Structured Data

**Sequence : The most basic data structure in Python where the index determines the order.**

ListTupleRangeCollections : Unordered data structures, hashable values.

**DictionariesSetsIterable** : Generic name for a sequence or collection; any object that can be iterated through. Can be mutable or immutable. Built In Data Types

---

**Lists are the python equivalent of arrays.**

```
1 empty_list = []
2 departments = ['HR', 'Development', 'Sales', 'Finance', 'IT', 'Customer Support']
```

**You can instantiate**

```
specials = list()
```

**Test if a value is in a list.**

```
1 print(1 in [1, 2, 3]) #> True
2 print(4 in [1, 2, 3]) #> False
3 # Tuples : Very similar to lists, but they are immutable
```

**Instantiated with parentheses**

```
time_blocks = ('AM', 'PM')
```

**Sometimes instantiated without**

```
1 colors = 'red','blue','green'
2 numbers = 1, 2, 3
```

## Tuple() built in can be used to convert other data into a tuple

```
1 tuple('abc') # returns ('a', 'b', 'c')
2 tuple([1,2,3]) # returns (1, 2, 3)
3 # Think of tuples as constant variables.
```

## Ranges : A list of numbers which can't be changed; often used with for loops.

Declared using one to three parameters.

Start : opt. default 0, first # in sequence. Stop : required next number past the last number in the sequence. Step : opt. default 1, difference between each number in the sequence.

```
1 range(5) # [0, 1, 2, 3, 4]
2 range(1,5) # [1, 2, 3, 4]
3 range(0, 25, 5) # [0, 5, 10, 15, 20]
4 range(0) # []
5 for let (i = 0; i < 5; i++)
6 for let (i = 1; i < 5; i++)
7 for let (i = 0; i < 25; i+=5)
8 for let(i = 0; i = 0; i++)
9 # Keep in mind that stop is not included in the range.
```

## Dictionaries : Mappable collection where a hashable value is used as a key to ref. an object stored in the dictionary.

Mutable.

```
1 a = {'one':1, 'two':2, 'three':3}
2 b = dict(one=1, two=2, three=3)
```

```
3 c = dict([('two', 2), ('one', 1), ('three', 3)])
4 # a, b, and c are all equal
```

### ***Declared with curly braces of the built in dict()***

Benefit of dictionaries in Python is that it doesn't matter how it is defined, if the keys and values are the same the dictionaries are considered equal.

**Use the in operator to see if a key exists in a dictionary.**

**Sets : Unordered collection of distinct objects; objects that need to be hashable.**

Always be unique, duplicate items are auto dropped from the set.

### **Common Uses:**

Removing DuplicatesMembership TestingMathematical Operators: Intersection, Union, Difference, Symmetric Difference.

**Standard Set is mutable, Python has a immutable version called frozenset. Sets created by putting comma seperated values inside braces:**

```
1 school_bag = {'book', 'paper', 'pencil', 'pencil', 'book', 'book', 'book', 'eraser'}
2 print(school_bag)
```

**Also can use set constructor to automatically put it into a set.**

```
1 letters = set('abracadabra')
2 print(letters)
3 #Built-In Functions
4 #Functions using iterables
```

**filter(function, iterable) : creates new iterable of the same type which includes each item for which the function returns true.**

**map(function, iterable) : creates new iterable of the same type which includes the result of calling the function on every item of the iterable.**

**sorted(iterable, key=None, reverse=False) : creates a new sorted list from the items in the iterable.**

**Output is always a list**

**key: opt function which converts and item to a value to be compared.**

**reverse: optional boolean.**

**enumerate(iterable, start=0) : starts with a sequence and converts it to a series of tuples**

```
1 quarters = ['First', 'Second', 'Third', 'Fourth']
2 print(enumerate(quarters))
3 print(enumerate(quarters, start=1))
```

**(0, 'First'), (1, 'Second'), (2, 'Third'), (3, 'Fourth')**

**(1, 'First'), (2, 'Second'), (3, 'Third'), (4, 'Fourth')**

**zip(\*iterables) : creates a zip object filled with tuples that combine 1 to 1 the items in each provided iterable.** Functions that analyze iterable

**len(iterable) : returns the count of the number of items.**

*\*max(args, key=None) : returns the largest of two or more arguments.*

**max(iterable, key=None) : returns the largest item in the iterable.**

*key optional function which converts an item to a value to be compared.* min works the same way as max

**sum(iterable) : used with a list of numbers to generate the total.**

*There is a faster way to concatenate an array of strings into one string, so do not use sum for that.*

**any(iterable) : returns True if any items in the iterable are true.**

**all(iterable) : returns True is all items in the iterable are true.**

---

## Working with dictionaries

**dir(dictionary) : returns the list of keys in the dictionary.** Working with sets

*\*Union : The pipe / operator or union(sets) function can be used to produce a new set which is a combination of all elements in the provided set.*

```
1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a | b) # => {1, 2, 3, 4, 6}
```

**Intersection : The & operator ca be used to produce a new set of only the elements that appear in all sets.**

```
1
2 a = {1, 2, 3}
3 b = {2, 4, 6}
4 print(a & b) # => {2}
5 Difference : The - operator can be used to produce a new set of only the elem
```

**Symmetric Difference : The ^ operator can be used to produce a new set of only the elements that appear in exactly one set and not in both.**

```
1 a = {1, 2, 3}
2 b = {2, 4, 6}
3 print(a - b) # => {1, 3}
4 print(b - a) # => {4, 6}
```

```
5 print(a ^ b) # => {1, 3, 4, 6}
```

## For StatementsIn python, there is only one for loop.

Always Includes:

1. The for keyword
2. A variable name
3. The 'in' keyword
4. An iterable of some kind
5. A colon
6. On the next line, an indented block of code called the for clause.

You can use break and continue statements inside for loops as well.

You can use the range function as the iterable for the for loop.

```
1 print('My name is')
2 for i in range(5):
3 print('Carlita Cinco (' + str(i) + ')')
4 total = 0
5 for num in range(101):
6 total += num
7 print(total)
8 Looping over a list in Python
9 for c in ['a', 'b', 'c']:
10 print(c)
11 lst = [0, 1, 2, 3]
12 for i in lst:
13 print(i)
```

*Common technique is to use the len() on a pre-defined list with a for loop to iterate over the indices of the list.*

```
1 supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
2 for i in range(len(supplies)):
3 print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
4
```

You can loop and destructure at the same time.

```
1 l = [1, 2], [3, 4], [5, 6]
2 for a, b in l:
3 print(a, ', ', b)
```

Prints 1, 2Prints 3, 4Prints 5, 6

**You can use values() and keys() to loop over dictionaries.**

```
1 spam = {'color': 'red', 'age': 42}
2 for v in spam.values():
3 print(v)
```

*Prints red*

*Prints 42*

```
1 for k in spam.keys():
2 print(k)
```

*Prints color*

*Prints age*

**For loops can also iterate over both keys and values.**

## Getting tuples

```
1 for i in spam.items():
2 print(i)
```

*Prints ('color', 'red')*

*Prints ('age', 42)*

*Destructuring to values*

```
1 for k, v in spam.items():
2 print('Key: ' + k + ' Value: ' + str(v))
```

*Prints Key: age Value: 42*

*Prints Key: color Value: red*

## Looping over string

```
1 for c in "abcdefg":
2 print(c)
```

**When you order arguments within a function or function call, the args need to occur in a particular order:**

*formal positional args.*

- args

*keyword args with default values*

- \*kwargs

```
1 def example(arg_1, arg_2, *args, **kwargs):
2 pass
3 def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blowfish", **kwargs)
4 pass
```

# Importing in Python

**Modules are similar to packages in Node.js** Come in different types:

Built-In,

Third-Party,

Custom.

**All loaded using import statements.**

---

## Terms

module : Python code in a separate file.package : Path to a directory that contains

modules.init.py : Default file for a package.submodule : Another file in a module's

folder.function : Function in a module.

**A module can be any file but it is usually created by placing a special file init.py into a folder.**

pic

*Try to avoid importing with wildcards in Python.*

*Use multiple lines for clarity when importing.*

```
1 from urllib.request import (
2 HTTPDefaultErrorHandler as ErrorHandler,
3 HTTPRedirectHandler as RedirectHandler,
4 Request,
5 pathname2url,
6 url2pathname,
7 urlopen,
8)
```

## Watching Out for Python 2

**Python 3 removed <> and only uses !=**

**format() was introduced with P3**

**All strings in P3 are unicode and encoded.md5 was removed.**

**ConfigParser was renamed to configparser** sets were killed in favor of set() class.

**print was a statement in P2, but is a function in P3.**

<https://gist.github.com/bgoonz/82154f50603f73826c27377ebaa498b5#file-python-study-guide-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

WK-18

# D4

Introduction to python taught through example problems. Solutions are included in embedded repl.it at the bottom of this page for you to...

---

## Python Problems & Solutions For Beginners

Introduction to python taught through example problems. Solutions are included in embedded repl.it at the bottom of this page for you to practice and refactor.

---

## Python Practice Problems

Here are some other articles for reference if you need them:

### Beginners Guide To Python

*My favorite language for maintainability is Python. It has simple, clean syntax, object encapsulation, good library...medium.com*

### Python Study Guide for a JavaScript Programmer

*A guide to commands in Python from what you know in JavaScript levelup.gitconnected.com*

---

Here are the problems without solutions for you to practice with:

<https://replit.com/@bgoonz/problems-witho-solutions>

---

## Problem 1

Create a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn `100` years old.

The `datetime` module supplies classes for manipulating dates and times.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

#### **[datetime - Basic date and time types - Python 3.9.6 documentation](#)**

*Only one concrete class, the class, is supplied by the module. The class can represent simple timezones with fixed...docs.python.org*

---

## **Problem 2**

Ask the user for a number. Depending on whether the number is `even` or `odd`, print out an appropriate message to the user.

### **Bonus:**

1. If the number is a multiple of `4`, print out a different message.
  2. Ask the user for two numbers: one number to check (call it `num`) and one number to divide by (`check`). If `check` divides evenly into `num`, tell that to the user. If not, print a different appropriate message.
- 

## **Problem 3**

Take a list and write a program that prints out all the elements of the list that are `less` than `5`.

### **Extras:**

1. Instead of printing the elements one by one, make a new list that has all the elements less than `5` from this list in it and print out this new list.
2. Write this in one line of Python.

- 
3. Ask the user for a number and return a list that contains only elements from the original list `a` that are smaller than that number given by the user.
- 

## Problem 4

Create a program that asks the user for a number and then prints out a list of all the divisors of that number. (If you don't know what a divisor is, it is a number that divides evenly into another number.)

For example, `13` is a divisor of `26` because `26 / 13` has no remainder.)

---

## Problem 5

Take two lists, and write a program that returns a list that contains only the elements that are common between the lists (without duplicates) . Make sure your program works on two lists of different sizes.

**random - Generate pseudo-random numbers - Python 3.9.6 documentation**

*Source code: Lib/random.py This module implements pseudo-random number generators for various distributions. For...docs.python.org*

Bonus:

1. Randomly generate two lists to test this.
  2. Write this in one line of Python.
- 

## Problem 6

Ask the user for a string and print out whether this string is a `palindrome` or not. (A palindrome is a string that reads the same forwards and backwards.)

Here's 5 ways to reverse a string (courtesy of [geeksforgeeks](#))

---

## Problem 7

Let's say I give you a list saved in a variable: `a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]` .

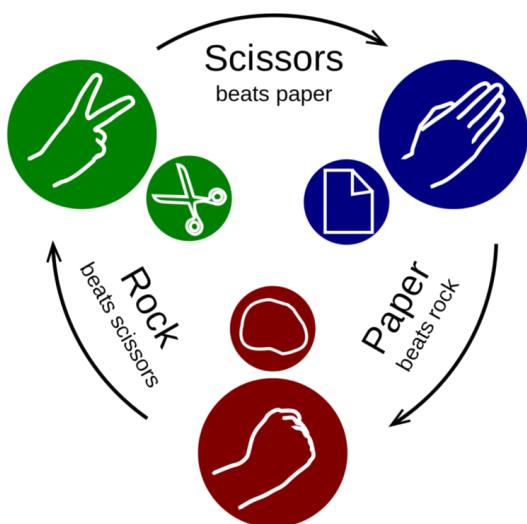
Write one line of Python that takes this list `a` and makes a new list that has only the even elements of this list in it.

---

## Problem 8

Make a two-player Rock-Paper-Scissors game.

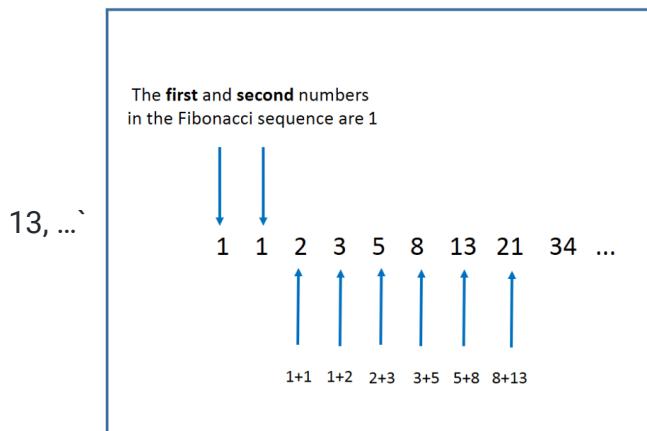
**Hint:** Ask for player plays (using `input`), compare them. Print out a message of congratulations to the winner, and ask if the players want to start a new game.



### Problem 9 Generate a random number between

`1 and 100 (including 1 and 100)`. Ask the user to guess the number, then tell them whether they guessed `too low`, `too high`, or `exactly right`. > \*\*Hint:\*\* Remember to use the user input from the very first exercise. \*\*Extras:\*\* Keep the game going until the user types ``exit``. Keep track of how many guesses the user has taken, and when the game ends, print this out. ### Problem 10 Write a program that asks the user how many Fibonacci numbers to generate and then generates them. Take this opportunity to think about how you can use functions. Make sure to ask the user to enter the number of numbers in the sequence to generate. \*\*Hint:\*\* The Fibonacci sequence is a sequence of numbers where the next number in the sequence is the

sum of the previous two numbers in the sequence. The sequence looks like this: `1, 1, 2, 3, 5, 8,



## Intermediate Problems:

### Problem 11

In linear algebra, a *Toeplitz matrix* is one in which the elements on any given diagonal from top left to bottom right are identical. Here is an example:

1	1	2	3	4	8
2	5	1	2	3	4
3	4	5	1	2	3
4	7	4	5	1	2

Write a program to determine whether a given input is a `Toeplitz` matrix.

### Problem 12

Given a positive integer `N`, find the smallest number of steps it will take to reach `1`.

There are two kinds of permitted steps: — → You may decrement N to N − 1. — → If  $a * b = N$ , you may decrement N to the larger of a and b.

For example, given 100, you can reach 1 in 5 steps with the following route:

100 → 10 → 9 → 3 → 2 → 1.

---

## Problem 13

Consider the following scenario: there are  $N$  mice and  $N$  holes placed at integer points along a line. Given this, find a method that maps mice to holes such that the largest number of steps any mouse takes is minimized.

Each move consists of moving one mouse one unit to the left or right, and only one mouse can fit inside each hole.

For example, suppose the mice are positioned at [1, 4, 9, 15], and the holes are located at [10, -5, 0, 16]. In this case, the best pairing would require us to send the mouse at 1 to the hole at -5, so our function should return 6.

---

## My Blog:

### Web-Dev-Hub

*Memoization, Tabulation, and Sorting Algorithms by Example Why is looking at runtime not a reliable method of...master-bgoonz-blog.netlify.app*

### A list of all of my articles to link to future posts

*You should probably skip this one... seriously it's just for internal use!bryanguner.medium.com*

# D3

## Objective 01 - Describe the properties of a binary tree and the properties of a "perfect" tree

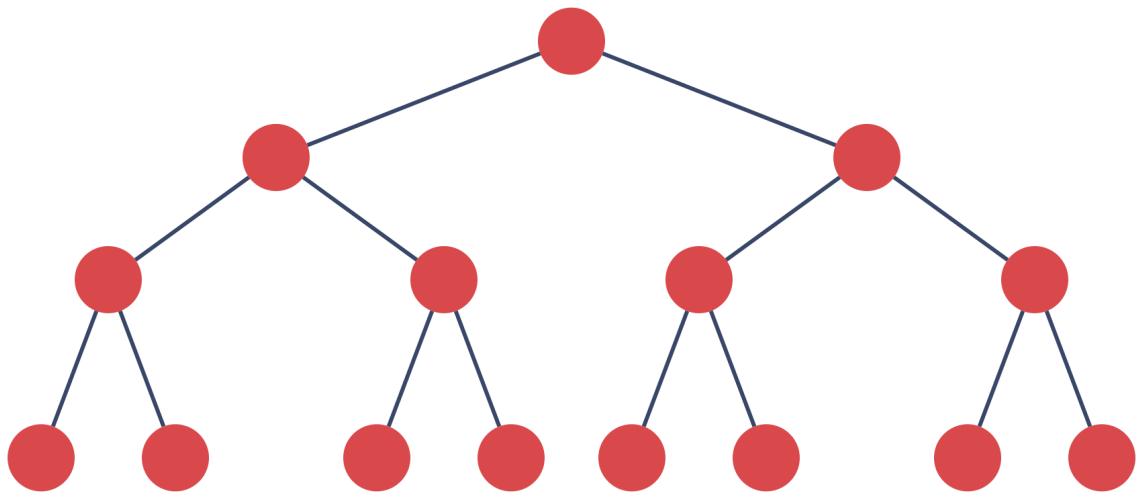
### Overview

There are lots of different types of tree data structures. A binary tree is a specific type of tree. It is called a binary tree because each node in the tree can only have a maximum of two child nodes. It is common for a node's children to be called either `left` or `right`. Here is an example of what a class for a binary tree node might look like:

```
1 class BinaryTreeNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
```

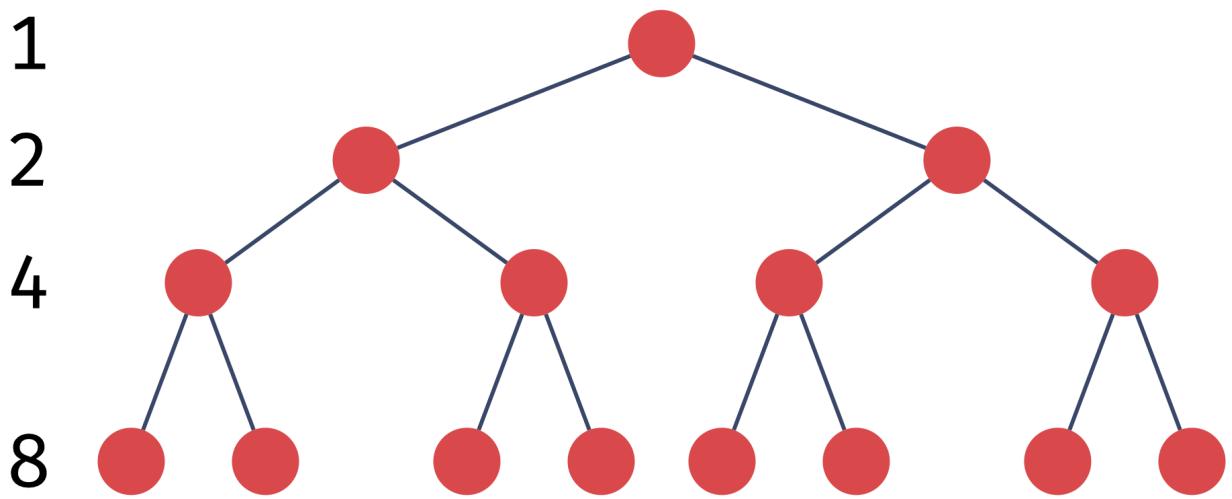
### Follow Along

With this simple class, we can now build up a structure that could be visualized like so:



[https://tk-assets.lambdaschool.com/c00c8f45-abff-4c3a-b29b-92631b5ac88e\\_binary-tree-example.001.png](https://tk-assets.lambdaschool.com/c00c8f45-abff-4c3a-b29b-92631b5ac88e_binary-tree-example.001.png)

### Perfect" Trees



[https://tk-assets.lambdaschool.com/36747e43-d96d-40c9-b8ab-d318f6da8aed\\_binary-tree-example-levels.001.png](https://tk-assets.lambdaschool.com/36747e43-d96d-40c9-b8ab-d318f6da8aed_binary-tree-example-levels.001.png)

A "perfect" tree has all of its levels full. This means that there are not any missing nodes in each level. "Perfect" trees have specific properties. First, the quantity of each level's nodes doubles as you go down. Second, the quantity of the last level's nodes is the same as the quantity of all the other nodes plus one. These properties are useful for understanding how to calculate the *height* of a tree. The height of a tree is the number of levels that it contains. Based on the properties outlined above, we can deduce that we can calculate the tree's height with the following formula:  $![\log_2(n+1) = h](\text{https://i.upmath.me/svg/log_2(n%2B1) %3D h})$  In the formula above,  $n$  is the total number of nodes. If you know the tree's height and want to calculate the total number of nodes, you can do so with the following formula:  $n = 2^h - 1$  We can represent the relationship between a perfect binary tree's total number of nodes and its height because of the properties outlined above.

## Challenge

1. Calculate how many levels a perfect binary tree has given that the total number of nodes is 127.
2. Calculate the total number of nodes on a perfect binary tree, given that the tree's height is 8.  
Additional Resources  
  3. [https://en.wikipedia.org/wiki/Binary\\_tree](https://en.wikipedia.org/wiki/Binary_tree) (Links to an external site.)
  4. <https://www.geeksforgeeks.org/binary-tree-data-structure/> (Links to an external site.)

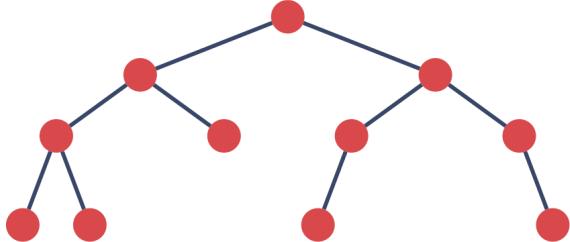
---

## Objective 02 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a binary search tree

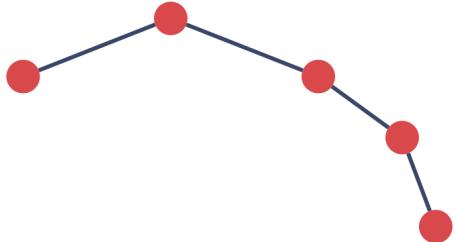
### Overview

Just like a binary tree is a specific type of tree, a binary search tree (BST) is a specific type of binary tree. A binary search tree is just like a binary tree, except it follows specific rules about how it orders the nodes contained within it. For each node in the BST, all the nodes to the left are smaller, and all the nodes to the right of it are larger. We can call a binary search tree balanced if the heights of its left and right subtrees differ by at most one, and both of the subtrees are also balanced.

**balanced**



**unbalanced**



[https://tk-assets.lambdaschool.com/f84f26b9-09f3-48e0-a4c6-a51740d9c083\\_binary-tree-example-balanced-unbalanced.001.png](https://tk-assets.lambdaschool.com/f84f26b9-09f3-48e0-a4c6-a51740d9c083_binary-tree-example-balanced-unbalanced.001.png)

## Follow Along

### Time and Space Complexity

#### Lookup

If a binary search tree is balanced, then a lookup operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is unbalanced, the time complexity can be linear ( $O(n)$ ) in the worst possible case (virtually a linear chain of nodes will have all the nodes on one side of the tree).

#### Insert

If a binary search tree is balanced, then an insertion operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is entirely unbalanced, then the time complexity is linear ( $O(n)$ ) in the worst case.

#### Delete

If a binary search tree is balanced, then a deletion operation's time complexity is logarithmic ( $O(\log n)$ ). If the tree is entirely unbalanced, then the time complexity is linear ( $O(n)$ ) in the

worst case.

## Space

The space complexity of a binary search tree is linear ( $O(n)$ ). Each node in the binary search tree will take up space in memory.

## Strengths

One of the main strengths of a BST is that it is sorted by default. You can pull out the data in order by using an in-order traversal. BSTs also have efficient searches ( $O(\log n)$ ). They have the same efficiency for their searches as a sorted array; however, BSTs are faster with insertions and deletions. In the average-case, dictionaries have more efficient operations than BSTs, but a BST has more efficient operations in the worst-case.

## Weaknesses

The primary weakness of a BST is that they only have efficient operations if they are balanced. The more unbalanced they are, the worse the efficiency of their operations gets. Another weakness is that they are don't have stellar efficiency in any one operation. They have good efficiency for a lot of different operations. So, they are more of a general-purpose data structure. If you want to learn more about trees that automatically rearrange their nodes to remain balanced, look into AVL trees ([Links to an external site.](#)) or Red-Black trees ([Links to an external site.](#))

## Challenge

1. In your own words, explain why an unbalanced binary search tree's performance becomes degraded.

Additional Resources

2. <https://www.geeksforgeeks.org/binary-search-tree-data-structure/> ([Links to an external site.](#))
3. [https://en.wikipedia.org/wiki/Binary\\_search\\_tree](https://en.wikipedia.org/wiki/Binary_search_tree) ([Links to an external site.](#))

---

## Objective 03 - Construct a binary search tree that can perform basic operations with a logarithmic time complexity

## Overview

To create a binary search tree, we need to define two different classes: one for the nodes that will make up the binary search tree and another for the tree itself.

## Follow Along

Let's start by creating a `BSTNode` class. An instance of `BSTNode` should have a `value`, a `right` node, and a `left` node.

```
1 class BSTNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
```

Now that we have our basic `BSTNode` class defined with an initialization method let's define our `BST` class. This class will have an initialization method and an `insert` method.

```
1 class BST:
2 def __init__(self, value):
3 self.root = BSTNode(value)
4 def insert(self, value):
5 self.root.insert(value)
```

Notice that our `BST` class expects each `BSTNode` to have an `insert` method available on an instance object. But, we haven't yet added an `insert` method on the `BSTNode` class. Let's do that now.

```
1 class BSTNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
6 def insert(self, value):
7 if value < self.value:
8 if self.left is None:
```

```
9 self.left = BSTNode(value)
10 else:
11 self.left.insert(value)
12 else:
13 if self.right is None:
14 self.right = BSTNode(value)
15 else:
16 self.right.insert(value)
```

Now that we can insert nodes into our binary search tree let's define a `search` method that can lookup values in our binary search tree.

```
1 class BST:
2 def __init__(self, value):
3 self.root = BSTNode(value)
4 def insert(self, value):
5 self.root.insert(value)
6 def search(self, value):
7 self.root.search(value)
```

Our `BST` class expects there to be a `search` method available on the `BSTNode` instance stored at the root. Let's go ahead and define that now.

```
1 class BSTNode:
2 def __init__(self, value):
3 self.value = value
4 self.left = None
5 self.right = None
6 def insert(self, value):
7 if value < self.value:
8 if self.left is None:
9 self.left = BSTNode(value)
10 else:
11 self.left.insert(value)
12 else:
13 if self.right is None:
14 self.right = BSTNode(value)
15 else:
16 self.right.insert(value)
17 def search(self, target):
18 if self.value == target:
19 return self
20 elif target < self.value:
```

```
21 if self.left is None:
22 return False
23 else:
24 return self.left.search(target)
25 else:
26 if self.right is None:
27 return False
28 else:
29 return self.right.search(target)
```

## Challenge

To implement a `delete` operation on our `BST` and `BSTNode` classes, we must consider three cases: 1. If the `BSTNode` to be deleted is a leaf (has no children), we can remove that node from the tree. 2. If the `BSTNode` to be deleted has only one child, we copy the child node to be deleted and delete it. 3. If the `BSTNode` to be deleted has two children, we have to find the "in-order successor". The "in-order successor" is the next highest value, the node that has the minimum value in the right subtree. Given the above information, can you write pseudocode for a method that can find the *minimum value* of all the nodes within a tree or subtree?

## Additional Resources

- <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/> (Links to an external site.)
- <https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/> (Links to an external site.)

# D2



qandstack.ipynb

<https://gist.github.com/bgoonz/4b9c322c9e2c303c0747e0cbf1e7d203#file-qandstack-ipynb>

## Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a queue

### Overview

A queue is a data structure that stores its items in a first-in, first-out (FIFO) order. That is precisely why it is called a queue. It functions just like a queue (or a line) would in everyday life. If you are the first to arrive at the check-in desk at a hotel, you will be the first to be served (and therefore, the first person to exit the queue). So, in other words, the items that are added to the queue first are the first items to be removed from the queue.

### Follow Along

#### Time and Space Complexity

##### Enqueue

To enqueue an item (add an item to the back of the queue) takes  $O(1)$  time.

##### Dequeue

To dequeue an item (remove an item from the front of the queue) takes  $O(1)$  time.

##### Peek

To peek at an item (inspect the item from the front of the queue without removing it) takes  $O(1)$  time.

## Space

The space complexity of a queue is linear ( $O(n)$ ). Each item in the queue will take up space in memory.

## Strengths

The primary strength of a queue is that all of its operations are fast (take  $O(1)$  time).

## Weaknesses

There are no weaknesses in this data structure. The reason is that it is a very targeted data structure designed to do a few things well.

## When are queues useful?

Queues are useful data structures in any situation where you want to make sure things are processed in a FIFO order. Think of a web server. The server might be trying to service thousands of page requests per minute. It would make the most sense for the server to process and respond to the requests in the same order that they were received. That way, the first client to request a page is the first client to receive a response. Also, you'll learn soon enough about traversing hierarchical data structures. One of the ways you do that is called a breadth-first traversal. To conduct a breadth-first traversal, a queue can be used.

## Challenge

1. In your own words, explain the strengths of a queue data structure.
2. If a queue only allows operations at the ends (front and back), what other data structure would be a perfect one to build the queue?

## Additional Resources

- <https://www.geeksforgeeks.org/queue-data-structure/> (Links to an external site.)

---

## Objective 02 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a stack

## Overview

A stack data structure handles information in a last-in, first-out order. This means that the last item added to the storage will be the first item removed from the storage. A stack is like having a paper tray inbox on your desk. Anytime a person walks by and drops a piece of paper or a letter in your inbox, it will go on the top of your inbox. So, when you process your inbox, the first item you would remove from the top of the stack of papers would be the last item added to it.

## Follow Along

### Time and Space Complexity

#### Push

To push an item (add an item to the top of the stack) takes  $O(1)$  time.

#### Pop

To pop an item (remove an item from the top of the stack) takes  $O(1)$  time.

#### Peek

To peek at an item (inspect the item from the top of the stack without removing it) takes  $O(1)$  time.

#### Space

The space complexity of a stack is linear ( $O(n)$ ). Each item in the stack will take up space in memory.

#### Strengths

The primary strength of a stack is that all of its operations are fast (take  $O(1)$  time).

#### Weaknesses

There are no weaknesses in this data structure. The reason is that it is a very targeted data structure designed to do a few things well.

## When are stacks useful?

Stacks can be useful in any situation where you desire a LIFO order. One common use-case is for parsing strings. Let's say you wanted to parse a string to ensure that all the parentheses in your string are correctly nested. A stack could be useful for this. When you encounter an opening parenthesis, you add it to the stack. When you encounter a closing parenthesis, you remove the top opening parenthesis from the stack. After going through all the characters in the string, the stack should be empty. If it isn't or if you try to remove an item from an empty stack, you'll know that the parentheses were not correctly nested.

Additionally, function calls and execution contexts are managed on a call stack. When you call a function, it's added to the call stack. When it returns, it gets popped off of the stack. Last, an iterative depth-first-search can be done using a stack.

## Challenge

1. In your own words, explain the strengths of a stack data structure.
2. What two data structures would work well for implementing a stack?

## Additional Resources

- <https://www.geeksforgeeks.org/stack-data-structure/> (Links to an external site.)
- 

# Objective 03 - Implement a queue using a linked list

## Overview

To implement a queue, we need to maintain two pointers. One pointer will point at the front (the first item) of the queue, and another pointer will point at the rear (the last item) of the queue.

Additionally, we need to have two methods available: `enqueue()` and `dequeue()`.

`enqueue()` adds a new item after the rear. `dequeue()` removes the front node and resets the front pointer to the next node.

## Follow Along

We will use a `LinkedListNode` class for each of the items in the queue.

```
1 class LinkedListNode:
2 def __init__(self, data):
3 self.data = data
4 self.next = None
```

For our `Queue` class, we first need to define an `__init__` method. This method should initialize our instance variables `front` and `rear`.

```
1 class Queue:
2 def __init__(self):
3 self.front = None
4 self.rear = None
```

Next, we need to define our `enqueue` method:

```
1 class Queue:
2 def __init__(self):
3 self.front = None
4 self.rear = None
5 def enqueue(self, item):
6 new_node = LinkedListNode(item)
7 # check if queue is empty
8 if self.rear is None:
9 self.front = new_node
10 self.rear = new_node
11 else:
12 # add new node to rear
13 self.rear.next = new_node
14 # reassign rear to new node
15 self.rear = new_node
```

Now, we need to define our `dequeue` method:

```
1 class Queue:
2 def __init__(self):
```

```
3 self.front = None
4 self.rear = None
5 def enqueue(self, item):
6 new_node = LinkedListNode(item)
7 # check if queue is empty
8 if self.rear is None:
9 self.front = new_node
10 self.rear = new_node
11 else:
12 # add new node to rear
13 self.rear.next = new_node
14 # reassign rear to new node
15 self.rear = new_node
16 def dequeue(self):
17 # check if queue is empty
18 if self.front is not None:
19 # keep copy of old front
20 old_front = self.front
21 # set new front
22 self.front = old_front.next
23
24 # check if the queue is now empty
25 if self.front is None:
26 # make sure rear is also None
27 self.rear = None
28
29 return old_front
```

Now we have a `Queue` class that uses a singly-linked list as the underlying data structure.

## Challenge

## Additional Resources

- <https://www.geeksforgeeks.org/queue-linked-list-implementation/>

---

## Objective 04 - Implement a stack using a dynamic array

### Overview

There are two common ways to implement a stack. One is by using a linked list, and the other is by using a dynamic array. Both of these implementations work well.

In the implementation that uses a dynamic array (a list in Python), the `push` method appends to the array, and the `pop` method removes the last element from the array.

## Follow Along

First we need to define our `Stack` class and define the `__init__` method:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
```

Now we need to define a `push` method to add an item to the top of our stack:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
4
5 def push(self, item):
6 self.data.append(item)
```

Next, we need to define a `pop` method to remove the top item from the stack:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
4
5 def push(self, item):
6 self.data.append(item)
7
8 def pop(self):
9 if len(self.data) > 0:
10 return self.data.pop()
11 return "The stack is empty"
```

## Challenge

## Additional Resources

- <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/> (Links to an external site.)

Immersive Reader

---

## Objective 04 - Implement a stack using a dynamic array

### Overview

There are two common ways to implement a stack. One is by using a linked list, and the other is by using a dynamic array. Both of these implementations work well.

In the implementation that uses a dynamic array (a list in Python), the `push` method appends to the array, and the `pop` method removes the last element from the array.

### Follow Along

First we need to define our `Stack` class and define the `__init__` method:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
```

Now we need to define a `push` method to add an item to the top of our stack:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
4
5 def push(self, item):
```

```
6 self.data.append(item)
```

Next, we need to define a `pop` method to remove the top item from the stack:

```
1 class Stack:
2 def __init__(self):
3 self.data = []
4
5 def push(self, item):
6 self.data.append(item)
7
8 def pop(self):
9 if len(self.data) > 0:
10 return self.data.pop()
11 return "The stack is empty"
```

## Challenge

## Additional Resources

- <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/> (Links to an external site.)

---

## Objective 05 - Implement a stack using a linked list

### Overview

There are two common ways to implement a stack. One is by using a linked list, and the other is by using a dynamic array. Both of these implementations work well.

In the implementation that uses a linked list, the `push` method inserts a new node at the linked list's head, and the `pop` method removes the node at the linked list's head.

### Follow Along

First, let's define our `Stack` class and its `__init__` method:

```
1 class LinkedListNode:
2 def __init__(self, data):
3 self.data = data
4 self.next = None
5
6 class Stack:
7 def __init__(self):
8 self.top = None
```

Now we need to define our `push` method to add items to the top of the stack.

```
1 class LinkedListNode:
2 def __init__(self, data):
3 self.data = data
4 self.next = None
5
6 class Stack:
7 def __init__(self):
8 self.top = None
9
10 def push(self, data):
11 # create new node with data
12 new_node = LinkedListNode(data)
13 # set current top to new node's next
14 new_node.next = self.top
15 # reset the top pointer to the new node
16 self.top = new_node
```

Next, we need to define our `pop` method to get items off the top of our stack.

```
1 class LinkedListNode:
2 def __init__(self, data):
3 self.data = data
4 self.next = None
5
6 class Stack:
7 def __init__(self):
8 self.top = None
9
10 def push(self, data):
```

```
11 # create new node with data
12 new_node = LinkedListNode(data)
13 # set current top to new node's next
14 new_node.next = self.top
15 # reset the top pointer to the new node
16 self.top = new_node
17
18 def pop(self):
19 # make sure stack is not empty
20 if self.top is not None:
21 # store popped node
22 popped_node = self.top
23 # reset top pointer to next node
24 self.top = popped_node.next
25 # return the value from the popped node
26 return popped_node.data
```

## Challenge

## Additional Resources

- <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/> (Links to an external site.)

# D1

## **Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a linked list**

### **Overview**

What is a linked list, and how is it different from an array? How efficient or inefficient are its operations? What are its strengths and weaknesses? How can I construct and interact with a linked list? By the end of this objective, you will be able to answer all of these questions confidently.

### **Follow Along**

#### **Basic Properties of a Linked List**

A linked list is a simple, linear data structure used to store a collection of elements. Unlike an array, each element in a linked list does not have to be stored contiguously in memory.

For example, in an array, each element of the list [43, 32, 63] is stored in memory like so:

43	32	63					
0	1	2	3	4	5	6	7

[https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767\\_arrays-stored-in-memory.001.jpeg](https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767_arrays-stored-in-memory.001.jpeg)

43 is the first item in the collection and is therefore stored in the first slot. 32 is the second item and is stored immediately next to 43 in memory. This pattern continues on and on.

In a linked list, each element of the list could be stored like so:

43	*3		32	*6		63	*None
0	1	2	3	4	5	6	7

[https://tk-assets.lambdaschool.com/72151497-7a5e-4940-835c-d8beb9c88922\\_linked-list-in-memory.001.jpeg](https://tk-assets.lambdaschool.com/72151497-7a5e-4940-835c-d8beb9c88922_linked-list-in-memory.001.jpeg)

You can see here that the elements can be spaced out in memory. Because the elements are not stored contiguously, each element in memory must contain information about the next element in the list. The first item stores the data `43` and the location in memory (`*3`) for the next item in the list. This example is simplified; the second item in the list `32` could be located anywhere in memory. It could even come before the first item in memory.

You might also be wondering what types of data can be stored in a linked list. Pretty much any type of data can be stored in a linked list. Strings, numbers, booleans, and other data structures can be stored. You should not feel limited using a linked list based on what type of data you are trying to store.

Are the elements in a linked list sorted or unsorted? The elements in a linked list can be either sorted or unsorted. There is nothing about the data structure that forces the elements to be sorted or unsorted. You cannot determine if a linked list's elements are sorted by determining they are stored in a linked list.

What about duplicates? Can a linked list contain them? Linked lists can contain duplicates. There is nothing about the linked list data structure that would prevent duplicates from being stored. When you encounter a linked list, you should know that it can contain duplicates.

Are there different types of linked lists? If so, what are they? There are three types of linked lists: singly linked list (SLL), doubly linked list (DLL), and circular linked list. All linked lists are made up of nodes where each node stores the data and also information about other nodes in the linked list.

Each singly linked list node stores the data and a pointer where the next node in the list is located. Because of this, you can only navigate in the forward direction in a singly linked list. To traverse an SLL, you need a reference to the first node called the head. From the head of the list, you can visit all the other nodes using the next pointers.

The difference between an SLL and a doubly linked list (DLL) is that each node in a DLL also stores a reference to the previous item. Because of this, you can navigate forward and backward in the list. A DLL also usually stores a pointer to the last item in the list (called the tail).

A Circular Linked List links the last node back to the first node in the list. This linkage causes a circular traversal; when you get to the end of the list, the next item will be back at the beginning of the list. Each type of linked list is similar but has small distinctions. When working with linked lists, it's essential to know what type of linked list.

## Time and Space Complexity

### Lookup

To look up an item by index in a linked list is linear time ( $O(n)$ ). To traverse through a linked list, you have to start with the head reference to the node and then follow each subsequent pointer to the next item in the chain. Because each item in the linked list is not stored contiguously in memory, you cannot access a specific index of the list using simple math. The distance in memory between one item and the next is varied and unknown.

### Append

Adding an item to a linked list is constant time ( $O(1)$ ). We always have a reference point to the tail of the linked list, so we can easily insert an item after the tail.

### Insert

In the worst case, inserting an item in a linked list is linear time ( $O(n)$ ). To insert an item at a specific index, we have to traverse — starting at the head — until we reach the desired index.

### Delete

In the worst case, deleting an item in a linked list is linear time ( $O(n)$ ). Just like insertion, deleting an item at a specific index means traversing the list starting at the head.

### Space

The space complexity of a linked list is linear ( $O(n)$ ). Each item in the linked list will take up space in memory.

## Strengths of a Linked List

The primary strength of a linked list is that operations on the linked list's ends are fast. This is because the linked list always has a reference to the head (the first node) and the tail (the last

node) of the list. Because it has a reference, doing anything on the ends is a constant time operation ( $O(1)$ ) no matter how many items are stored in the linked list. Additionally, just like a dynamic array, you don't have to set a capacity to a linked list when you instantiate it. If you don't know the size of the data you are storing, or if the amount of data is likely to fluctuate, linked lists can work well. One benefit over a dynamic array is that you don't have doubling appends. This is because each item doesn't have to be stored contiguously; whenever you add an item, you need to find an open spot in memory to hold the next node.

## **Weaknesses of a Linked List**

The main weakness of a linked list is not efficiently accessing an "index" in the middle of the list. The only way that the linked list can get to the seventh item in the linked list is by going to the head node and then traversing one node at a time until you arrive at the seventh node. You can't do simple math and jump from the first item to the seventh.

## **What data structures are built on linked lists?**

Remember that linked lists have efficient operations on the ends (head and tail). There are two structures that only operate on the ends; queues and stacks. So, most queue or stack implementations use a linked list as their underlying data structure.

## **Why is a linked list different than an array? What problem does it solve?**

We can see the difference between how a linked list and an array are stored in memory, but why is this important? Once you see the problem with the way arrays are stored in memory, the benefits of a linked list become clearer.

The primary problem with arrays is that they hold data contiguously in memory. Remember that having the data stored contiguously is the feature that gives them quick lookups. If I know where the first item is stored, I can use simple math to figure out where the fifth item is stored. The reason that this is a problem is that it means that when you create an array, you either have to know how much space in memory you need to set aside, or you have to set aside a bunch of extra memory that you might not need, just in case you do need it. In other words, you can be space-efficient by only setting aside the memory you need at the moment. But, in doing that, you are setting yourself up for low time efficiency if you run out of room and need to copy all of your elements to a newer, bigger array.

With a linked list, the elements are not stored side-by-side in memory. Each element can be stored anywhere in memory. In addition to storing the data for that element, each element also

stores a pointer to the memory location of the next element in the list. The elements in a linked list do not have an index. To get to a specific element in a linked list, you have to start at the head of the linked list and work your way through the list, one element at a time, to reach the specific element you are searching for. Now you can see how a linked list solves some of the problems that the array data structure has.

### How do you represent a linked list graphically and in Python code?

Let's look at how we can represent a singly linked list graphically and in Python code. Seeing a singly linked list represented graphically and in code can help you understand it better.

How do you represent a singly linked list graphically? Let's say you wanted to store the numbers 1, 2, and 3. You would need to create three nodes. Then, each of these nodes would be linked together using the pointers.



[https://tk-assets.lambdaschool.com/baa6486b-9322-481e-95be-c660640c4966\\_linked-list-graphical-representation.001.jpeg](https://tk-assets.lambdaschool.com/baa6486b-9322-481e-95be-c660640c4966_linked-list-graphical-representation.001.jpeg)

Notice that the last element or node in the linked list does not have a pointer to any other node. This fact is how you know you are at the end of the linked list.

What does a singly linked list implementation look like in Python? Let's start by writing a `LinkedListNode` class for each element in the linked list.

```
1 class LinkedListNode:
2 def __init__(self, data=None, next=None):
3 self.data = data
4 self.next = next
```

Now, we need to build out the class for the `LinkedList` itself:

```
1 class LinkedList:
2 def __init__(self, head=None):
3 self.head = head
```

Our class is super simple so far and only includes an initialization method. Let's add an `append` method so that we can add nodes to the end of our list:

```
1 class LinkedList:
2 def __init__(self, head=None):
3 self.head = head
4
5 def append(self, data):
6 new_node = LinkedListNode(data)
7
8 if self.head:
9 current = self.head
10
11 while current.next:
12 current = current.next
13
14 current.next = new_node
15 else:
16 self.head = new_node
```

Now, let's use our simple class definitions for `LinkedListNode` and `LinkedList` to create a linked list of elements `1`, `2`, and `3`.

```
1 >>> a = LinkedListNode(1)
2 >>> my_ll = LinkedList(a)
3 >>> my_ll.append(2)
4 >>> my_ll.append(3)
```

```
5 >>> my_ll.head.data
6 1
7 >>> my_ll.head.next.data
8 2
9 >>> my_ll.head.next.next.data
10 3
11 >>>
```

You must be able to understand and interact with linked lists. You now know the basic properties and types of linked lists, what makes a linked list different from an array, what problem it solves, and how to represent them both graphically and in code. You now know enough about linked lists that you should be able to solve algorithmic code challenges that require a basic understanding of linked lists.

## Challenge

1. Draw out a model of a singly-linked list that stores the following integers in order:

3,2,6,5,7,9 .

2. Draw out a model of a doubly-linked list that stores the following integers in order:

5,2,6,4,7,8 .

## Additional Resources

- <https://www.cs.cmu.edu/~fp/courses/15122-f15/lectures/10-linkedlist.pdf> (Links to an external site.)
- [https://www.youtube.com/watch?v=njTh\\_OwMljA](https://www.youtube.com/watch?v=njTh_OwMljA) (Links to an external site.)



```
1 # -*- coding: utf-8 -*-
2 """Linked Lists.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at
7 https://colab.research.google.com/drive/17MD2e14fi7n95HTvy1K_ttM0FZSYnLmm
8
9 # Linked Lists
```

```
10 - Non Contiguous abstract Data Structure
11 - Value (can be any value for our use we will just use numbers)
12 - Next (A pointer or reference to the next node in the list)
```

L1 = Node(34) L1.next = Node(45) L1.next.next = Node(90)

---

**while the current node is not none**

---

**do something with the data**

---

**traverse to next node**

L1 = [34]->[45]->[90] -> None

Node(45) Node(90)

```
1 """
2
3 class LinkedListNode:
4 """
5 Simple Singly Linked List Node Class
6 value -> int
7 next -> LinkedListNode
8 """
9 def __init__(self, value):
10 self.value = value
11 self.next = None
12
13 def add_node(self, value):
14 # set current as a ref to self
15 current = self
16 # while there is still more nodes
17 while current.next is not None:
18 # traverse to the next node
19 current = current.next
20 # create a new node and set the ref from current.next to the new node
```

```
21 current.next = LinkedListNode(value)
22
23 def insert_node(self, value, target):
24 # create a new node with the value provided
25 new_node = LinkedListNode(value)
26 # set a ref to the current node
27 current = self
28 # while the current nodes value is not the target
29 while current.value != target:
30 # traverse to the next node
31 current = current.next
32 # set the new nodes next pointer to point toward the current nodes next po
33 new_node.next = current.next
34 # set the current nodes next to point to the new node
35 current.next = new_node
36
37 ll_storage = []
38 L1 = LinkedListNode(34)
39 L1.next = LinkedListNode(45)
40 L1.next.next = LinkedListNode(90)
41
42 def print_ll(linked_list_node):
43 current = linked_list_node
44 while current is not None:
45 print(current.value)
46 current = current.next
47
48 def add_to_ll_storage(linked_list_node):
49 current = linked_list_node
50 while current is not None:
51 ll_storage.append(current)
52 current = current.next
53
54 L1.add_node(12)
55 print_ll(L1)
56 L1.add_node(24)
57 print()
58 print_ll(L1)
59 print()
60 L1.add_node(102)
61 print_ll(L1)
62 L1.insert_node(123, 90)
63 print()
64 print_ll(L1)
65 L1.insert_node(678, 34)
66 print()
67 print_ll(L1)
68 L1.insert_node(999, 102)
69 print()
70 print_ll(L1)
71
72 """# CODE 9571"""
73
```

```

74 class LinkedListNode:
75 """
76 Simple Doubly Linked List Node Class
77 value -> int
78 next -> LinkedListNode
79 prev -> LinkedListNode
80 """
81 def __init__(self, value):
82 self.value = value
83 self.next = None
84 self.prev = None
85
86 """
87 Given a reference to the head node of a singly-linked list, write a function
88 that reverses the linked list in place. The function should return the new head
89 of the reversed list.
90 In order to do this in O(1) space (in-place), you cannot make a new list, you
91 need to use the existing nodes.
92 In order to do this in O(n) time, you should only have to traverse the list
93 once.
94 *Note: If you get stuck, try drawing a picture of a small linked list and
95 running your function by hand. Does it actually work? Also, don't forget to
96 consider edge cases (like a list with only 1 or 0 elements).*
97 cn p
98 None [1] -> [2] ->[3] -> None
99
100
101 - setup a current variable pointing to the head of the list
102 - set up a prev variable pointing to None
103 - set up a next variable pointing to None
104
105 - while the current ref is not none
106 - set next to the current.next
107 - set the current.next to prev
108 - set prev to current
109 - set current to next
110
111 - return prev
112
113 """
114 class LinkedListNode():
115 def __init__(self, value):
116 self.value = value
117 self.next = None
118
119 def reverse(head_of_list):
120 current = head_of_list
121 prev = None
122 next = None
123
124 while current:
125 next = current.next
126 current.next = prev

```

```

127 prev = current
128 current = next
129
130 return prev
131
132 class HashTableEntry:
133 """
134 Linked List hash table key/value pair
135 """
136 def __init__(self, key, value):
137 self.key = key
138 self.value = value
139 self.next = None
140
141
142 # Hash table can't have fewer than this many slots
143 MIN_CAPACITY = 8
144
145 [
146 0["Lou", 41] -> ["Bob", 41] -> None,
147 1["Steve", 41] -> None,
148 2["Jen", 41] -> None,
149 3["Dave", 41] -> None,
150 4None,
151 5["Hector", 34]-> None,
152 6["Lisa", 41] -> None,
153 7None,
154 8None,
155 9None
156]
157 class HashTable:
158 """
159 A hash table that with `capacity` buckets
160 that accepts string keys
161 Implement this.
162 """
163
164 def __init__(self, capacity):
165 self.capacity = capacity # Number of buckets in the hash tab
166 self.storage = [None] * capacity
167 self.item_count = 0
168
169
170 def get_num_slots(self):
171 """
172 Return the length of the list you're using to hold the hash
173 table data. (Not the number of items stored in the hash table,
174 but the number of slots in the main list.)
175 One of the tests relies on this.
176 Implement this.
177 """
178 # Your code here
179

```

```
180
181 def get_load_factor(self):
182 """
183 Return the load factor for this hash table.
184 Implement this.
185 """
186 return len(self.storage)
187
188
189 def djb2(self, key):
190 """
191 DJB2 hash, 32-bit
192 Implement this, and/or FNV-1.
193 """
194 str_key = str(key).encode()
195
196 hash = FNV_offset_basis_64
197
198 for b in str_key:
199 hash *= FNV_prime_64
200 hash ^= b
201 hash &= 0xffffffffffffffff # 64-bit hash
202
203 return hash
204
205
206 def hash_index(self, key):
207 """
208 Take an arbitrary key and return a valid integer index
209 between within the storage capacity of the hash table.
210 """
211 return self.djb2(key) % self.capacity
212
213 def put(self, key, value):
214 """
215 Store the value with the given key.
216 Hash collisions should be handled with Linked List Chaining.
217 Implement this.
218 """
219 index = self.hash_index(key)
220
221 current_entry = self.storage[index]
222
223 while current_entry is not None and current_entry.key != key:
224 current_entry = current_entry.next
225
226 if current_entry is not None:
227 current_entry.value = value
228 else:
229 new_entry = HashTableEntry(key, value)
230 new_entry.next = self.storage[index]
231 self.storage[index] = new_entry
232
```

```
233
234 def delete(self, key):
235 """
236 Remove the value stored with the given key.
237 Print a warning if the key is not found.
238 Implement this.
239 """
240 # Your code here
241
242
243 def get(self, key):
244 """
245 Retrieve the value stored with the given key.
246 Returns None if the key is not found.
247 Implement this.
248 """
249 # Your code here
```