

datastructures-in-python

Home

self link:

<https://bgoonz42.gitbook.io/datastructures-in-python/>

Python Practice Files:

 [Python Prac Zip File](#)

examples.zip - 132KB

Main Repo:

 [GitHub - bgoonz/DATA_STRUC_PYTHON_NOTES](#)

https://github.com/bgoonz/DATA_STRUC_PYTHON_NOTES

Website:

 [Python Notes](#)

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

 [ds-algo \(forked\) - CodeSandbox](#)

<https://codesandbox.io/s/ds-algo-forked-e754i>

The Algorithms Reference Site:

 [DS-Algo-Codebase](#)

<https://bgoonz-branch-the-algos.vercel.app/>

Notion:

 [Notion – The all-in-one workspace for your notes, tasks, wikis, and databases.](#)

<https://www.notion.so/webdevhub42/Python-Data-Structures-Unit-1da9a5d55db844f4b62aff6fd2b4d1ce>

Data Structures & Algorithm Interview Codebase (mostly JS):



[GitHub - bgoonz/Data-Structures-Algos-Codebase](https://github.com/bgoonz/Data-Structures-Algos-Codebase)

<https://github.com/bgoonz/Data-Structures-Algos-Codebase>

Global Site Tag:

```
1  <!-- Global site tag (gtag.js) - Google Analytics -->
2  <script async src="https://www.googletagmanager.com/gtag/js?id=G-4HQK8JZK1T"></script>
3  <script>
4    window.dataLayer = window.dataLayer || [];
5    function gtag(){dataLayer.push(arguments);}
6    gtag('js', new Date());
7
8    gtag('config', 'G-4HQK8JZK1T');
9  </script>
```

Cirriculum

Outline

🔗 iframe inception

<https://60s1b.csb.app/>

→ wk17

</cirriculum/untitled-3>

→ wk18

</cirriculum/untitled-2>

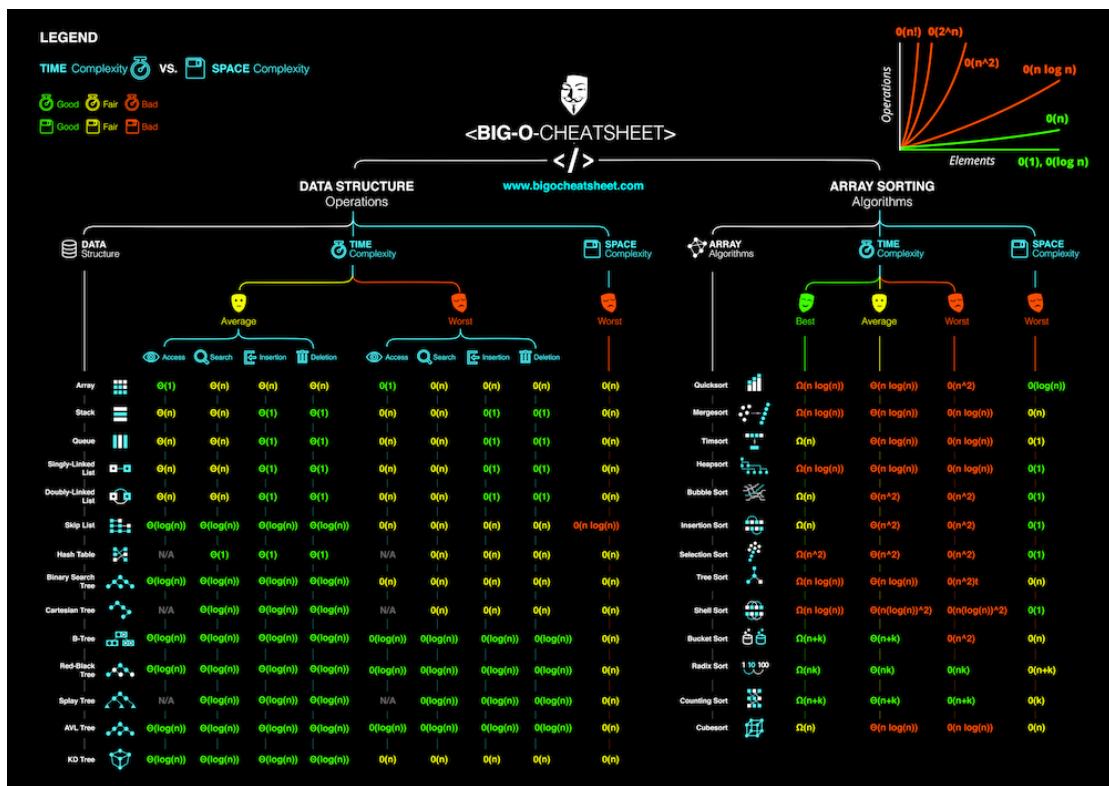
→ wk19

</cirriculum/untitled-4>

→ wk20

</cirriculum/untitled-1>

wk17



→ Outline

/cirriculum/untitled

→ D1-Module 01 - Python I

/cirriculum/untitled-3/untitled-2

→ D2- Module 02 - Hash Tables I

/cirriculum/untitled-2/untitled-2

→ D3- Module 03 - Python III

/cirriculum/untitled-3/untitled-1

→ D4- Module 04 - Searching and Recursion

/cirriculum/untitled-2/untitled

→ wk18

/cirriculum/untitled-2

→ wk19

/cirriculum/untitled-4

→ wk20

/cirriculum/untitled-1

Outline

Overview

During this sprint, we will introduce you to some fundamental Computer Science fundamentals. First, we will introduce you to Python (the language we will be using throughout Computer Science). Second, you will learn about Lambda's Problem-Solving Framework, which we call U.P.E.R. Third, you will learn about Big O notation and analyzing an algorithm's time and space complexity. Last, you will learn about arrays and in-place and out-of-place algorithms.

All of these topics lay down a crucial base for the other three sprints in Computer Science. You will rely on your Python skills, problem-solving abilities, ability to analyze time and space complexity, and your mental model for computer memory throughout the rest of the course.

Python I

In this module, you will begin to learn the fundamentals of the Python programming language. Additionally, you will learn about the U.P.E.R. Problem-Solving framework and best practices for asking for help. After completing this module, you will have all the basics that you need to get started using Python to solve algorithmic code challenges and deepen your understanding and skill set related to programming.

Python II

In this module, you will continue to learn the fundamentals of the Python programming language and put the U.P.E.R. Problem-Solving framework into practice. These skills are crucial as you encounter harder and harder code challenges in preparation for the technical interview process.

Python III

This module will teach about mutability vs. immutability, time complexity analysis, and space complexity analysis. These topics are incredibly crucial for optimizing the algorithms that you write and ensuring that we prepare you for the technical interview process.

Python IV

In this module, you will learn about static arrays, dynamic arrays, and the difference between an in-place algorithm and an out-of-place algorithm.

homework

Create a function that returns `True` if the given string has any of the following:

- Only letters and no numbers.
- Only numbers and no letters.

If a string has both numbers and letters or contains characters that don't fit into any category, return `False`.

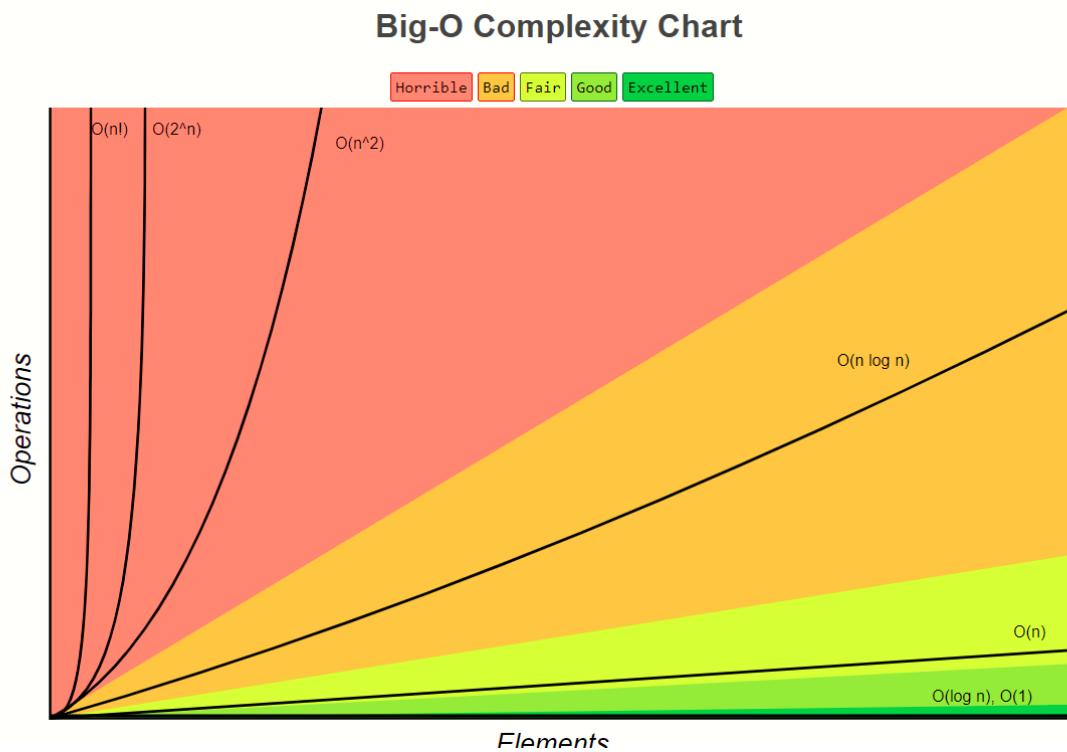
Examples:

- `csAlphanumericRestriction("Bold") → True`
- `csAlphanumericRestriction("123454321") → True`
- `csAlphanumericRestriction("H3LL0") → False`

Notes:

- Any string that contains spaces or is empty should return `False`.
- **[execution time limit] 4 seconds (py3)**
- **[input] string input_str**
- **[output] boolean**

D1-Module 01 - Python I



 d1

<https://replit.com/@bgoonz/d1#main.py>

Windows

Windows machines usually do not ship with Python installed. Installing on Windows is pretty simple.

1. Download the latest Python 3 Installer from python.org ([Links to an external site.](#)) (make sure you pay attention to 32-bit vs. 64-bit and select the right one for your machine).
2. Run the installer and **make sure you check the box that says "Add Python 3.x to PATH"** to ensure that you place the interpreter in your execution path.

Linux

Most likely, your Linux distribution already has Python installed. However, it is likely to be Python 2 and not Python 3.

You can determine what version you have by opening a terminal and typing `python --version`. If the version shown is `Python 2.x.x`, then you want to install the latest version of Python 3.

The procedure for installing the latest version of Python depends on which distribution of Linux you are running.

Use [this article](#) ([Links to an external site.](#)) to find instructions specific to your Linux distribution.

macOS / Mac OS X

Current versions of macOS include a version of Python 2, but you want to be using Python 3.

The best way to install Python 3 on macOS is to use the Homebrew package manager.

Install Homebrew

1. Go to <http://brew.sh/> ([Links to an external site.](#)) and select the Homebrew bootstrap code under "Install Homebrew" and copy the complete command to your clipboard.
 2. Open a terminal window, paste the Homebrew bootstrap code, and hit "Enter."
 3. It may take some time to install Homebrew, so you need to wait for that process to complete before moving on.

After Homebrew has finished its installation process, you then need to install Python.

Install Python

1. Open a terminal and run the following command `brew install python3`. This command will download and install the latest version of Python.
 2. Ensure that everything was installed correctly by opening a terminal window and running the command `pip3`.
 3. If you see help text from Python's "pip" package manager, you have a working Python installation.

Online Interpreters

Here are a few websites that give you online access to the Python interpreter:

- Repl.it (Links to an external site.)
 - Trinket (Links to an external site.)
 - Python Fiddle (Links to an external site.)
 - Python.org Online Console (Links to an external site.)
 - Python Anywhere



Search and Research

Before you do anything else, search for a solution to your problem on your own. One thing you should start doing is keeping track of all your research when solving a problem. One easy way to do this is to have a browser window represent a specific search for a solution, and each open tab represents an attempt at solving it. Keeping track of your research is vital because it's helpful to provide examples of similar questions or similar problems and explain why those didn't answer your specific problem or question. It also helps the person answering your question avoid pointing you toward resources you've already explored, and lets them know that you've already put in the work.

Introduce the Problem

The first thing you do when you ask a question is to introduce the problem. The first paragraph of your written question should serve as an executive summary of the problem. All the following paragraphs should fill in the details of the problem.

An important thing to include in your problem introduction is a precise explanation of how you encountered the problem. Write about the difficulties that kept you from solving it. Describe what you already tried and include the results of the research you've done.

You should also provide as much detail about the context as possible. For instance, include the language version, the platform version, the operating system, the database type, specific IDE, and any web server information. You should also include your particular constraints. For example, you may not be allowed to use feature A or B that would provide an obvious solution. If you have an odd constraint, it may also help explain why you have that constraint.

Help Others Reproduce the Problem

One thing to remember is that not all questions benefit from including code. However, if you include code, definitely do not just copy in your entire program! By having irrelevant details, you make your question much harder to answer and decrease the chances of someone helping you.

Here are some guidelines for when to include code in your question.

Minimal

Include just enough code to allow others to reproduce your specific problem. One way to do this is to restart from scratch. Do not include snippets of your entire program. Instead, create a new program, but only add what's necessary to recreate the problem.

If you aren't exactly sure where the problem code is, one way to find it is by removing code chunks one at a time until the problem disappears – then add back the last part. This way, you can deduce that the last piece of code you added back is likely the source of your problem.

Be careful not to remove too much code, either. Keep your question brief, but maintain enough context for clarity.

Complete

Make sure you include all the portions of the code needed to reproduce the problem. It would be best if you assumed that the person who is answering your question would not write any code to reproduce your issue. Again, remember, do not use images of code—those trying to help you need direct access to the code you include in your question.

Reproducible

When you include your code, it's also important to tell the reader exactly what you expect the behavior to be. Be sure to show the reader the exact wording of the error message you encountered (if there was one). It's also crucial to double-check that your included example reproduces the problem.

One other thing you can do is create a live example on a site like [sqlfiddle.com](#) or [jsbin.com](#). If you do, make sure you also include a copy of your code in your question. Not everyone will utilize the link to the live example.

And to reiterate, do not post images of any code, data, or error messages—reserve images for things like rendering bugs—things that are impossible to describe accurately with just text.

Proofread

Don't send a question you haven't proofread. When you post your question, you should have already read and reread it, taking care to follow all the best practices and making sure your question makes sense. It would be best if you imagined that you're coming to your question fresh, with no other context but the question itself. You want to make your question as easy for someone to answer as possible. Remember, the reader is likely choosing between several questions they could answer. You want your question to stand out as something concise and approachable. Don't forget to double-check your spelling, grammar, and formatting. Keep it as straightforward as you can; you're not writing a novel.

Respond to Feedback

As feedback and responses to your question begin coming in, respond as quickly as possible. You'll likely receive clarifying questions, and your readers need that clarification to help you.

Follow Along

Now let's look at an example of a question posted to Stack Overflow and analyze it to see if it follows the best practices outlined above.

The question ([Links to an external site.](#)):

Accessing the index in 'for' loops?

Asked 11 years, 1 month ago Active 1 month ago Viewed 2.0m times

How do I access the index in a `for` loop like the following?

3494

```
ints = [8, 23, 45, 12, 78]
for i in ints:
    print('item #{}) = {}'.format(???, i))
```

696

I want to get this output:

⌚

```
item #1 = 8
item #2 = 23
item #3 = 45
item #4 = 12
item #5 = 78
```

When I loop through it using a `for` loop, how do I access the loop index, from 1 to 5 in this case?

`python` `loops` `list`

<https://camo.githubusercontent.com/9be35d94fd27e59fc716b00942c22b5b3438a99a/68747470733a2f2f746b2d6173736574732e6c16d6264617363686f6f6c2e636f6d2f64383031393630662d626530662d346633362d383634612d356462666616435306631635f53637265656e53686f74323032302d30332d33306174332e31352e3330504d2e706e67>

The first thing to notice is that the post has a short but descriptive title that adequately summarizes the question.

Accessing the index in 'for' loops?

Next, did the questioner provide any additional context or proof of the research they've done so far? It doesn't look like it. They could improve the question by including what they tried and the resources they explored.

The questioner did an excellent job of introducing the question and including code that shows what they are trying to do. In this case, they did not need to include experience vs. expected behavior; they just needed to have the expected behavior. By clearly stating what the desired result was, it helped the person answering to respond appropriately.

The code they included is a minimal and complete example, allowing someone to reproduce the problem quickly. The questioner left out irrelevant details and code that would've distracted from the primary question. They also included an example of what the desired output would be, which is helpful.

It appears the questioner proofread their question beforehand as it does not contain any glaring spelling, grammar, or formatting problems. However, we could critique this example for including a redundant sentence at the end. Instead of including that sentence, they might have rephrased the first sentence of the question to be more precise.

Challenge

1. Choose a real-world example from a recent problem/challenge. Use the guidelines and process outlined above to ask for help in your cohort-specific help channel.
2. Identify an unanswered question in your cohort-specific help channel. Do your best to provide a helpful response to that question.
3. Find an example of a **bad** question on Stack Overflow. Analyze the question using the guidelines above and write a short response explaining *why* you believe it is a **bad** question.
4. Find an example of a **good** question on Stack Overflow. Analyze the question using the guidelines above and write a short response explaining *why* you believe it is a **good** question.

Additional Resources

- Stack Overflow: How Do I Ask a Good Question? ([Links to an external site.](#))
- Writing the Perfect Question ([Links to an external site.](#))
- How to Ask Questions the Smart Way ([Links to an external site.](#))
- How to Debug Small Programs ([Links to an external site.](#))



Objective 04 - Use a print statement

Overview

Learning to use the `print` function in Python is the perfect way to start writing Python code. When learning to write in any new programming language, one of the first things you want to do is get some *output* from your program. The `print` function is how you output the value of an object to the screen. You will learn how to use the `print` function in Python.

Follow Along

Using `print` with different objects

Let's start by executing the `print` function to print different types of objects in Python. There are numerous types of objects that you can print using the `print` function.

Using `print` with no arguments:

```
1 >>> print()  
2  
3 >>>
```

Notice the empty line after calling the `print` function. The default `end` value when calling `print` is the newline character `\n`.

Using `print` with a string literal:

```
1 >>> print("Lambda School is awesome!")  
2 Lambda School is awesome!  
3 >>>
```

Notice how calling `print` with the string literal printed the exact string we passed in onto the screen.

Using `print` with a variable:

```
1 >>> slogan = "i love lamp"  
2 >>> print(slogan)  
3 i love lamp  
4 >>>
```

Notice how calling `print` with the `slogan` variable prints the value assigned to the `slogan` variable.

Using `print` with an expression:

```
1 >>> superlative = "wonderful"
2 >>> school = "Lambda School"
3 >>> print(school + " is " + superlative)
4 Lambda School is wonderful
5 >>>
```

Notice how the argument for the `print` function can be an expression. Once the expression is resolved to a string object, the `print` function can output it to the screen.

Using `print` with other object types:

```
1 print(2020)
2 2020
3 >>> print(123.456)
4 123.456
5 >>> print(False)
6 False
7 >>> print(["Lambda", "School", 2, 0, 2, 0])
8 ['Lambda', 'School', 2, 0, 2, 0]
9 >>> print(("Lambda", "School"))
10 ('Lambda', 'School')
11 >>> print({"school": "Lambda School", "year": 2020})
12 {'school': 'Lambda School', 'year': 2020}
13 >>>
```

Any object passed as an argument into `print` will get converted into a string type before outputted to the screen.

You can see how the `print` function is easy to use and how it can handle any object type that you pass into it.

Passing multiple arguments into `print`

Now, let's look at how we can pass multiple arguments into the `print` function. Using `print` with multiple arguments gives you a flexible and easy way to output items to the screen.

We can pass multiple objects, all of the same or different types, into `print`.

```
1 >>> print("Lambda School", 2020, True)
2 Lambda School 2020 True
3 >>>
```

Notice how each object we passed in was converted to a string and then output to the screen. Notice also that `print` used `" "` as the default separator value.

We can change the separator value by assigning a value to the keyword argument `sep`.

```
1 >>> print("Lambda School", 2020, True, sep="!!!")
2 Lambda School!!!2020!!!True
3 >>> print("Lambda School", 2020, True, sep="\t")
4 Lambda School    2020      True
5 >>> print("Lambda School", 2020, True, sep="\n")
6 Lambda School
7 2020
8 True
```

```
9 >>> print("Lambda School", 2020, True, sep="")
10 Lambda School2020True
11 >>>
```

Specifying the `end` value with `print`

You can also specify the `end` value by assigning a value to the `end` keyword argument when you call the `print` function. Being able to print a value to the screen but allow the user to stay on the same line is useful and necessary in some cases.

Here is how you can change the default `end` value (which is `\n`) when calling the `print` function.

```
1 >>> print("Are you a Lambda School student?", end=" (Y or N)")
2 Are you a Lambda School student? (Y or N)>>>
```

Customizing the `end` value when calling the `print` function can be useful and necessary in some circumstances.

You have now learned the basics of using the `print` function in Python. You learned how to call the `print` function to print objects of different types. You now know how to use `print` with multiple positional arguments. In certain necessary situations, you also know how to change the default `end` value when calling the `print` function.

Now, get some practice using the `print` function by completing the challenge below.



cs-unit-1-sprint-1-module-1-print-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-print-3#main.py>



Objective 04 - Use a `print` statement

Overview

Learning to use the `print` function in Python is the perfect way to start writing Python code. When learning to write in any new programming language, one of the first things you want to do is get some *output* from your program. The `print` function is how you output the value of an object to the screen. You will learn how to use the `print` function in Python.

Follow Along

Using `print` with different objects

Let's start by executing the `print` function to print different types of objects in Python. There are numerous types of objects that you can print using the `print` function.

Using `print` with no arguments:

```
1 >>> print()  
2  
3 >>>
```

Notice the empty line after calling the `print` function. The default `end` value when calling `print` is the newline character `\n`.

Using `print` with a string literal:

```
1 >>> print("Lambda School is awesome!")  
2 Lambda School is awesome!  
3 >>>
```

Notice how calling `print` with the string literal printed the exact string we passed in onto the screen.

Using `print` with a variable:

```
1 >>> slogan = "i love lamp"  
2 >>> print(slogan)  
3 i love lamp  
4 >>>
```

Notice how calling `print` with the `slogan` variable prints the value assigned to the `slogan` variable.

Using `print` with an expression:

```
1 >>> superlative = "wonderful"  
2 >>> school = "Lambda School"  
3 >>> print(school + " is " + superlative)  
4 Lambda School is wonderful  
5 >>>
```

Notice how the argument for the `print` function can be an expression. Once the expression is resolved to a string object, the `print` function can output it to the screen.

Using `print` with other object types:

```
1 print(2020)  
2 2020  
3 >>> print(123.456)  
4 123.456  
5 >>> print(False)  
6 False  
7 >>> print(["Lambda", "School", 2, 0, 2, 0])  
8 ['Lambda', 'School', 2, 0, 2, 0]  
9 >>> print(("Lambda", "School"))  
10 ('Lambda', 'School')  
11 >>> print({"school": "Lambda School", "year": 2020})  
12 {'school': 'Lambda School', 'year': 2020}  
13 >>>
```

Any object passed as an argument into `print` will get converted into a string type before outputted to the screen.

You can see how the `print` function is easy to use and how it can handle any object type that you pass into it.

Passing multiple arguments into `print`

Now, let's look at how we can pass multiple arguments into the `print` function. Using `print` with multiple arguments gives you a flexible and easy way to output items to the screen.

We can pass multiple objects, all of the same or different types, into `print`.

```
1 >>> print("Lambda School", 2020, True)
2 Lambda School 2020 True
3 >>>
```

Notice how each object we passed in was converted to a string and then output to the screen. Notice also that `print` used `" "` as the default separator value.

We can change the separator value by assigning a value to the keyword argument `sep`.

```
1 >>> print("Lambda School", 2020, True, sep="!!!")
2 Lambda School!!!2020!!!True
3 >>> print("Lambda School", 2020, True, sep="\t")
4 Lambda School    2020      True
5 >>> print("Lambda School", 2020, True, sep="\n")
6 Lambda School
7 2020
8 True
9 >>> print("Lambda School", 2020, True, sep="")
10 Lambda School2020True
11 >>>
```

Specifying the `end` value with `print`

You can also specify the `end` value by assigning a value to the `end` keyword argument when you call the `print` function. Being able to print a value to the screen but allow the user to stay on the same line is useful and necessary in some cases.

Here is how you can change the default `end` value (which is `\n`) when calling the `print` function.

```
1 >>> print("Are you a Lambda School student?", end=" (Y or N)")
2 Are you a Lambda School student? (Y or N)>>>
```

Customizing the `end` value when calling the `print` function can be useful and necessary in some circumstances.

You have now learned the basics of using the `print` function in Python. You learned how to call the `print` function to print objects of different types. You now know how to use `print` with multiple positional arguments. In certain necessary situations, you also know how to change the default `end` value when calling the `print` function.

Now, get some practice using the `print` function by completing the challenge below.

Challenge

Additional Resources

- https://www.w3schools.com/python/ref_func_print.asp



Objective 05 - Use white space to denote blocks

Overview

Python is unique because indentation instead of some other character marks blocks of code. A block of code is a collection of statements that are grouped. The syntax for denoting blocks varies from language to language. For example, in C, blocks are delimited by curly braces ({ }). Understanding how Python uses whitespace and indentation to denote logical lines and code blocks is essential.

Follow Along

Whitespace Characters

Whitespace is any character represented by something that appears empty (usually `\t` or `" "`). The characters that Python considers to be whitespace can be seen by printing out the value of `string.whitespace` from the `string` library.

```
1 >>> import string  
2 >>> string.whitespace  
3 '\t\n\r\x0b\x0c'  
4 >>>
```

Notice the characters are " " (space), \t (tab), \n (newline), \r (return), \x0b (unicode line tabulation), and \x0c (unicode form feed).

You've seen the different types of whitespace characters that can appear, but you mainly need to concern yourself with " " , \t , and \n .

Logical Lines of Code

Whitespace is used to denote the end of a logical line of code. In Python, a logical line of code's end (a statement or a definition) is marked by a `\n`.

```
1 >>> first = "Lambda"
2 >>> second = "School"
3 >>> first + second
4 'LambdaSchool'
5 >>> first \
6 ... + \
7 ... second
8 'LambdaSchool'
9 >>>
```

Notice how the REPL evaluates the expression `first + second` when I return on line 3. Below that, I can write one logical line of code over multiple lines by ending each line with a `\` character. That `\` character lets the Python interpreter know that even though there is a newline you don't want it to treat it as the end of a logical line.

It's important to understand that Python assumes meaning in newline characters when trying to interpret your code.

Code Blocks

Whitespace (indentation) can denote code blocks. Python gives meaning to the amount of whitespace (indentation level) that comes before a logical line of code.

```
1 >>> if True:  
2 ...     if True:  
3     File "<stdin>", line 2  
4         if True:  
5             ^  
6IndentationError: expected an indented block  
7 >>>
```

This code raises an `Indentation Error` because the Python interpreter expects to find additional whitespace inside the `if` block.

```
1 >>> if True:  
2 ...     if True:  
3 ...         print("it worked!")  
4 ...  
5 it worked!  
6 >>>
```

The Python interpreter can successfully run this code because consistent whitespace (level of indentation) is used.

```
1 >>> if True:  
2 ...     if True:  
3 ...         print("it worked!")  
4 File "<stdin>", line 3  
5     print("it worked!")  
6             ^  
7 TabError: inconsistent use of tabs and spaces in indentation
```

Although you can't tell in the code snippet above, for the second `if` statement, I used a `\t` to indent. But, for the indentation on `print("it worked!")`, I used eight `" "` (spaces). The mismatch of tab usage and spaces raises an error when Python tries to interpret the code.

Consistent whitespace usage (indentation) is crucial to making sure that Python can interpret your code correctly.

In Python, whitespace has meaning; it denotes the end of logical lines and also code blocks. Whitespace is any character represented by something that appears empty, although the most common characters are `" "`, `\t`, and `\n`. The Python interpreter knows where the end of a logical line of code is because of the `\n`. The amount of whitespace (level of indentation) is used in Python to denote blocks of code. Understanding how the Python interpreter looks at whitespace is vital to writing valid Python code.



cs-unit-1-sprint-1-module-1-white-space-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-white-space-3#main.py>



#6:

Overview

Python is not a "statically typed" language, and every variable in Python is an object. You don't have to declare a variable's type.

Follow Along

Numbers

In Python, you can have integers and floating-point numbers.

You can define an integer like so:

```
my_int = 3
```

You can also cast a floating-point number to be an integer like so:

```
my_int = int(3.0)
```

To define a floating-point number, you can declare it literally or typecast it with the float constructor function:

```
1 my_float = 3.0
2 my_float = float(3)
```

Strings

You can define strings with either single or double quotes:

```
1 my_string = 'Lambda School'
2 my_string = "Lambda School"
```

It's common to use double quotes for strings so that you can include apostrophes without accidentally terminating the string.

```
my_string = "I don't have to worry about apostrophes with my double-quotes."
```

Let's practice declaring variables to store an int, a float, and a string:

```
1 my_int = 2
2 my_float = 5.0
3 my_str = "Lambda School"
```



cs-unit-1-sprint-1-module-1-basic-types-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-types-3>



Overview

There are a few basic operators that you should be familiar with as you start writing Python code.

Arithmetic Operators

You can use the addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) operators with numbers in Python.

```
1 my_number = 2 + 2 * 8 / 5.0
2 print(my_number) # 5.2
```

There is also an operator called the modulo operator (`%`). This operator returns the remainder of integer division.

```
1 my_remainder = 9 % 4
2 print(my_remainder) # 1
```

You can use two multiplication operators to make the exponentiation operator (`**`).

```
1 two_squared = 2 ** 2
2 print(two_squared)    # 4
3 two_cubed = 2 ** 3
4 print(two_cubed)      # 8
```

Using operators with non-numbers

You can use the addition operator to concatenate strings and lists:

```
1 string_one = "Hello,"
2 string_two = " World!"
3 combined = string_one + string_two
4 print(combined) # Hello, World!
5
6 lst_one = [1,2,3]
7 lst_two = [4,5,6]
8 big_lst = lst_one + lst_two
9 print(big_lst) # [1, 2, 3, 4, 5, 6]
```

You can also use the multiplication operator to create a new list or string that repeats the original sequence:

```
1 my_string = "Bueller"
```

```
2 repeated = my_string * 3
3 print(repeated) # BuellerBuellerBueller
4
5 my_list = [1, 2, 3]
6 repeated_list = my_list * 3
7 print(repeated_list) # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Follow Along

Now, let's see if we can combine all of this information in a quick demo.

First, let's create two variables, `a` and `b`, where each variable stores an instance of the `object` class.

```
1 a = object()
2 b = object()
```

Next, let's see if we can make two lists, one containing five instances of `a`, and the second with five instances of `b`.

```
1 a_list = [a] * 5
2 b_list = [b] * 5
```

Then, let's combine `a_list` and `b_list` into a `combined` list.

```
combined = a_list + b_list
```

If our code works as expected, `combined` should have a length of 10.

```
print(len(combined)) # 10
```



cs-unit-1-sprint-1-module-1-basic-operators-1

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-operators-1#main.py>



Overview

To format a string in Python, you use the `%` operator to format a set of stored variables in a tuple. You also include *argument specifiers* in your string with special symbols like `%s` and `%d`.

For example, let's say you want to insert a `name` variable inside a string. You would do the following:

```
1 name = "Austen"
2 formatted_string = "Hello, %s!" % name
3 print(formatted_string) # Hello, Austen!
```

If you have more than one argument specifier, you need to enclose your arguments in a tuple:

```
1 name = "Austen"
2 year = 2020
3 print("Hey %s! It's the year %d." % (name, year))
4 # Hey Austen! It's the year 2020.
```

Any object that is not a string can also be formatted using the `%s` operator. The string which returns from the object's `repr` method will be used in the formatted string.

```
1 my_list = [1,2,3]
2 print("my_list: %s" % my_list)
3 # my_list: [1, 2, 3]
```

A few of the common argument specifiers are:

- `%s` - String (or any object with a string representation)
- `%d` - Integers
- `%f` - Floating point numbers
- `%.<number of digits>f` - Floating point numbers with a fixed amount of digits to the dot's right.
- `%x/%X` - Integers in hexadecimal (lowercase/uppercase)

Follow Along

Let's see if we can use all of this information to practice formatting a few strings.

Let's imagine that we have some data that we want to inject into a string.

```
1 product_name = "bananas"
2 price = 1.23
3 product_id = 123456
```

We need to print a formatted string using argument specifiers and a tuple that contains our data:

```
1 print("%s (id: %d) are currently $%.2f." % (product_name, product_id, price))
2 # bananas (id: 123456) are currently $1.23.
```





8

Overview

You can think of a string as anything between quotes. Strings store a sequence of characters or bits of text.

There are lots of ways you can interact with strings in Python.

Follow Along

The `len()` method prints out the number of characters in the string.

```
1 my_string = "Hello, world!"  
2 print(len(my_string)) # 12
```

The `index()` method prints out the index of the substring argument's first occurrence.

```
1 my_string = "Hello, world!"  
2 print(my_string.index("o")) # 4  
3 print(my_string.index(", w")) # 5
```

The `count()` method returns the number of occurrences of the substring argument.

```
1 my_string = "Hello, world!"  
2 print(my_string.count("o")) # 2  
3 print(my_string.count("ll")) # 1
```

To slice a string, you can use this syntax: `[start:stop:step]`. To reverse the string's order, you can set the step value to be `-1`.

```
1 my_string = "Hello, world!"  
2 print(my_string[3:7]) # lo,  
3 print(my_string[3:7:2]) # l,  
4 print(my_string[::-1]) # !dlrow ,olleH
```

You can convert a string to uppercase or lowercase with the `upper()` and `lower()` methods.

```
1 my_string = "Hello, world!"  
2 print(my_string.upper()) # HELLO, WORLD!  
3 print(my_string.lower()) # hello, world!
```

You can determine if a string starts with or ends with a specific sequence with the `startswith()` and `endswith()` methods.

```
1 my_string = "Hello, world!"  
2 print(my_string.startswith("Hello")) # True  
3 print(my_string.endswith("globe!")) # False
```

The `split()` method allows you to split up a string into a list. The default separator is any whitespace. You can also specify the separator value with an argument if you want.

```
1 my_string = "Hello, world!"  
2 print(my_string.split()) # ['Hello,', 'world!']  
3 print(my_string.split(",")) # ['Hello', ' world!']  
4 print(my_string.split("l")) # ['He', '', 'o, wor', 'd!']
```



cs-unit-1-sprint-1-module-1-basic-string-operations-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-string-operations-2>



Objective 09 - Perform basic string operations

Overview

You can think of a string as anything between quotes. Strings store a sequence of characters or bits of text.

There are lots of ways you can interact with strings in Python.

Follow Along

The `len()` method prints out the number of characters in the string.

```
1 my_string = "Hello, world!"  
2 print(len(my_string)) # 12
```

The `index()` method prints out the index of the substring argument's first occurrence.

```
1 my_string = "Hello, world!"  
2 print(my_string.index("o")) # 4  
3 print(my_string.index("l", w")) # 5
```

The `count()` method returns the number of occurrences of the substring argument.

```
1 my_string = "Hello, world!"  
2 print(my_string.count("o")) # 2  
3 print(my_string.count("ll")) # 1
```

To slice a string, you can use this syntax: `[start:stop:step]`. To reverse the string's order, you can set the step value to be `-1`.

```
1 my_string = "Hello, world!"  
2 print(my_string[3:7]) # lo,  
3 print(my_string[3:7:2]) # l,  
4 print(my_string[::-1]) # !dlrow ,olleH
```

You can convert a string to uppercase or lowercase with the `upper()` and `lower()` methods.

```
1 my_string = "Hello, world!"  
2 print(my_string.upper()) # HELLO, WORLD!  
3 print(my_string.lower()) # hello, world!
```

You can determine if a string starts with or ends with a specific sequence with the `startswith()` and `endswith()` methods.

```
1 my_string = "Hello, world!"  
2 print(my_string.startswith("Hello")) # True  
3 print(my_string.endswith("globe!")) # False
```

The `split()` method allows you to split up a string into a list. The default separator is any whitespace. You can also specify the separator value with an argument if you want.

```
1 my_string = "Hello, world!"  
2 print(my_string.split()) # ['Hello,', 'world!']  
3 print(my_string.split(",")) # ['Hello', ' world!']  
4 print(my_string.split("l")) # ['He', '', 'o, wor', 'd!']  
5
```



cs-unit-1-sprint-1-module-1-basic-string-operations-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-basic-string-operations-3>



Overview

Python uses boolean values to evaluate conditions. An expression in any Boolean context will evaluate to a Boolean value and then control your program's flow. Python's boolean values are written as `True` and `False` (make sure you capitalize the first character).

Follow Along

To compare the value of two expressions for equality, you use the `==` operator. You can also use `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), and `!=` (not equal).

```
1 x = 10
2 print(x == 10) # True
3 print(x == 5) # False
4 print(x < 15) # True
5 print(x > 15) # False
6 print(x <= 10) # True
7 print(x >= 10) # True
8 print(x != 20) # True
```

You build up more complex boolean expressions by using the `and` and `or` operators.

```
1 name = "Elon"
2 age = 49
3 if name == "Elon" and age == 49:
4     print("You are a 49 year old person named Elon.")
5
6 if name == "Elon" or name == "Bill":
7     print("Your name is either Elon or Bill.")
```

Any time you have an iterable object (like a list), you can check if a specific item exists inside that iterable by using the `in` operator.

```
1 years = [2018, 2019, 2020, 2021]
2 year = 2020
3
4 if year in years:
5     print("%s is in the years collection" % year)
6
7 # 2020 is in the years collection
```

We can use the `if`, `elif`, and the `else` keywords to define a series of code blocks that will execute conditionally.

```
1 first_statement = False
2 second_statement = True
3
4 if first_statement:
5     print("The first statement is true")
6 elif second_statement:
7     print("The second statement is true")
8 else:
9     print("Neither the first statement nor the second statement are true")
```

Any object that is considered "empty" evaluates to `False`. For example, `""`, `[]`, and `0` all evaluate to `False`.

If we want to determine if two objects are actually the same instance in memory, we use the `is` operator instead of the value comparison operator `==`.

```
1 a = [1,2,3]
2 b = [1,2,3]
3
4 print(a == b) # True because a and b have the same value
5 print(a is b) # False because a and b reference two different list objects
6
7 x = [1,2,3]
8 y = x
9
10 print(x == y) # True because x and y have the same value
11 print(x is y) # True because x and y reference the same list object
```

There is also the `not` operator, which inverts the boolean that follows it:

```
1 print(not False)    # True
2 print(not (1 == 1)) # False because 1 == 1 is True and then is inverted by not
```



cs-unit-1-sprint-1-module-1-conditional-expressions-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-conditional-expressions-2#main.py>



Overview

You can use two types of loops in Python, a `for` loop and a `while` loop. A `for` loop iterates over a given sequence (iterator expression). A `while` loop repeats as long as a boolean context evaluates to `True`.

The `break` statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the `break` statement is inside a nested loop (loop inside another loop), the `break` statement will only terminate the innermost loop.

You can use the `continue` statement to skip the rest of the code inside a loop *for the current iteration only*. The loop does not terminate entirely but continues with the next iteration.

Follow Along

Here is an example of a few different ways you can use a `range` as the iterable for a `for` loop.

```
1 # Prints 0, 1, 2, 3, 4
2 for x in range(5):
3     print(x)
4
5 # Prints 2, 3, 4, 5, 6
6 for x in range(2, 7):
7     print(x)
8
9 # Prints 1, 3, 5, 7
10 for x in range(1, 8, 2):
11     print(x)
```

This example shows the simple usage of a `while` loop to print the same values as the `for` loops above.

```
1 # Prints 0, 1, 2, 3, 4
2 count = 0
3 while count < 5:
4     print(count)
5     count += 1
6
7 # Prints 2, 3, 4, 5, 6
8 count = 2
9 while count < 7:
10    print(count)
11    count += 1
12
13 # Prints 1, 3, 5, 7
14 count = 1
15 while count < 8:
16    print(count)
17    count += 2
```

You can use a `break` statement to exit a `for` loop or a `while` loop.

```
1 # Prints 0, 1, 2, 3, 4
2 count = 0
3 while True:
4     print(count)
5     count += 1
6     if count >= 5:
7         break
```

You can also use a `continue` statement to skip the current block but not exit the loop entirely.

```
1 # Prints 1, 3, 5, 7
2 for x in range(8):
3     # if x is even, skip this block and do not print
4     if x % 2 == 0:
5         continue
6     print(x)
```



cs-unit-1-sprint-1-module-1-loops-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-loops-2#main.py>



Objective 12 - Create user-defined functions and call them

To make our code more readable and DRY (Don't Repeat Yourself), we often want to encapsulate code inside a callable function.

To define a function in Python, we follow this syntax:

```
1 def function_name(argument_1, argument_2, etc.):
2     # function line 1
3     # function line 2
4     # etc.
```

Follow Along

Let's define a greeting function that allows us to specify a name and a specific greeting.

```
1 def greet(name, greeting):
2     print("Hello, %s, %s" % (name, greeting))
```

Now, we can call our `greet` function and pass in the data that we want.

```
1 greet("Austen", "I hope you are having an excellent day!")
2 # Hello, Austen, I hope you are having an excellent day!
```

If we want to define a function that returns a value to the caller, we use the `return` keyword.

```
1 def double(x):
2     return x * 2
3
4 eight = double(4)
5 print(eight)
6 # 8
```



 cs-unit-1-sprint-1-module-1-loops-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-loops-3#main.py>

 intro-2-python-w-jupyter

<https://gist.github.com/bgoonz/4f5c0b5fe80a84421ff9a5a66dc e29da>

Overview

Python uses boolean values to evaluate conditions. An expression in any Boolean context will evaluate to a Boolean value and then control your program's flow. Python's boolean values are written as `True` and `False` (make sure you capitalize the first character).

Follow Along

To compare the value of two expressions for equality, you use the `==` operator. You can also use `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), and `!=` (not equal).

```
1 x = 10
2 print(x == 10) # True
3 print(x == 5) # False
4 print(x < 15) # True
5 print(x > 15) # False
6 print(x <= 10) # True
7 print(x >= 10) # True
8 print(x != 20) # True
```

You build up more complex boolean expressions by using the `and` and `or` operators.

```
1 name = "Elon"
2 age = 49
3 if name == "Elon" and age == 49:
4     print("You are a 49 year old person named Elon.")
5
6 if name == "Elon" or name == "Bill":
7     print("Your name is either Elon or Bill.")
```

Any time you have an iterable object (like a list), you can check if a specific item exists inside that iterable by using the `in` operator.

```
1 years = [2018, 2019, 2020, 2021]
2 year = 2020
3
4 if year in years:
5     print("%s is in the years collection" % year)
6
7 # 2020 is in the years collection
```

We can use the `if`, `elif`, and the `else` keywords to define a series of code blocks that will execute conditionally.

```
1 first_statement = False
2 second_statement = True
3
4 if first_statement:
5     print("The first statement is true")
6 elif second_statement:
7     print("The second statement is true")
8 else:
9     print("Neither the first statement nor the second statement are true")
```

Any object that is considered "empty" evaluates to `False`. For example, `" "`, `[]`, and `0` all evaluate to `False`.

If we want to determine if two objects are actually the same instance in memory, we use the `is` operator instead of the value comparison operator `==`.

```
1 a = [1,2,3]
2 b = [1,2,3]
3
4 print(a == b) # True because a and b have the same value
5 print(a is b) # False because a and b reference two different list objects
6
7 x = [1,2,3]
8 y = x
9
10 print(x == y) # True because x and y have the same value
11 print(x is y) # True because x and y reference the same list object
```

There is also the `not` operator, which inverts the boolean that follows it:

```
1 print(not False)      # True
2 print(not (1 == 1)) # False because 1 == 1 is True and then is inverted by not
```

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

D2- Module 02 - Python II



[GitHub - wilfredinni/python-cheatsheet: Basic Cheat Sheet for Python \(PDF, Markdown and Jupyter Notebook\)](https://github.com/wilfredinni/python-cheatsheet)

<https://github.com/wilfredinni/python-cheatsheet>



[Number Bases and Chars.ipynb](https://gist.github.com/bgoonz/ebe842b651bba237d70bccfa5a7b5a75#file-number-bases-and-chars-ipynb)

<https://gist.github.com/bgoonz/ebe842b651bba237d70bccfa5a7b5a75#file-number-bases-and-chars-ipynb>

Overview

A module is a collection of code that is written to meet specific needs. For example, you could split up different parts of a game you were building into modules. Each module would be a separate Python file that you could manage separately.

Follow Along

Any Python file that ends with the `.py` extension is considered a module. The name of the module is the name of the file.

To import from other modules, we can use the `import` command.

```
1 import math
2
3 print(math.factorial(5))
4 # 120
```

So, by importing the built-in `math` module, we have access to all of the functions and data defined in that module. We access those functions and data using dot notation, just like we do with objects.

If you only need a specific function from a module, you can import that specific function like so:

```
1 from math import factorial
2
3 print(factorial(5))
4 # 120
```

You can also import all the names from a module with this syntax to avoid using dot notation throughout your file.

```
1 from math import *
2
3 print(factorial(5))
4 # 120
5 print(pow(2, 3))
6 # 8.0
```

You can also bind the module to a name of your choice by using `as`.

```
1 import math as alias
2
3 print(alias.factorial(5))
4 # 120
```

To find out which names a module defines when imported, you can use the `dir()` method. This method returns an alphabetically sorted list of strings for all of the names defined in the module.

```
1 import math
2
3 print(dir(math))
4 # ['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', '
```

Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a linked list

<https://youtu.be/PC0w44UH7Mo>

<https://replit.com/@bgoonz/Comments-1>

<https://gist.github.com/bgoonz/73035b719d10a753a44089b41eacf6ca#file-copy-of-linked-lists-ipynb>

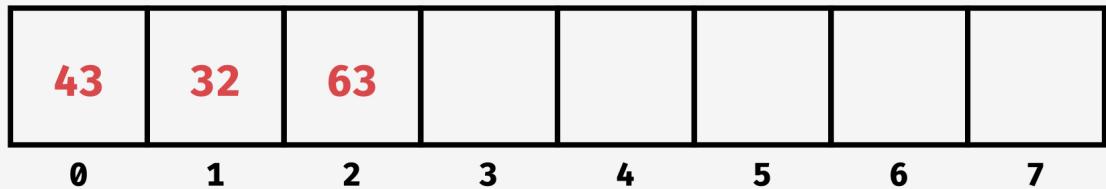
What is a linked list, and how is it different from an array? How efficient or inefficient are its operations? What are its strengths and weaknesses? How can I construct and interact with a linked list? By the end of this objective, you will be able to answer all of these questions confidently.

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.

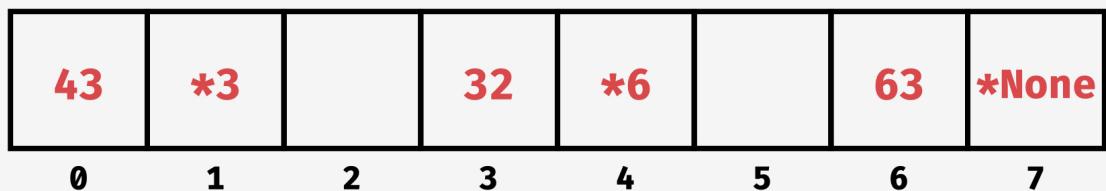
For example, in an array, each element of the list `[43, 32, 63]` is stored in memory like so:



https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767_arrays-stored-in-memory.001.jpeg

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

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.

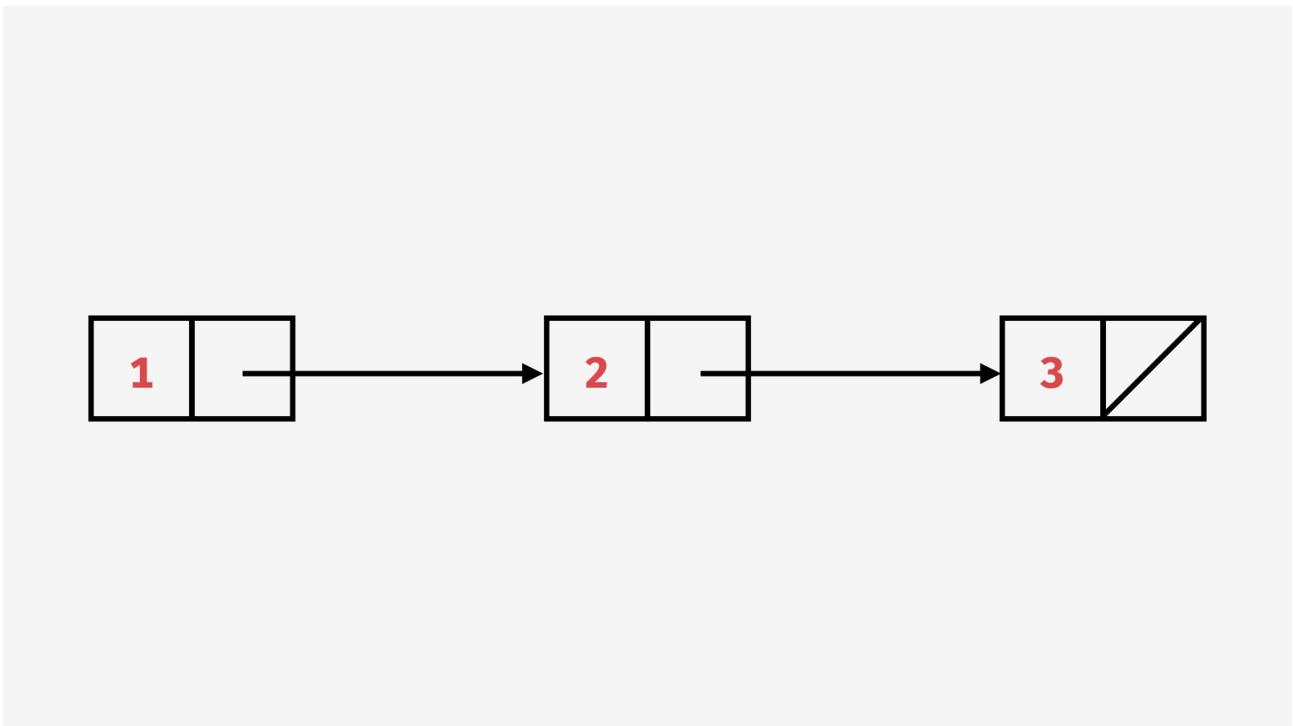
The primary problem with arrays is that they hold data contiguously in memory. Remember that having the data stored contiguously is the feature that gives them quick lookups. If I know where the first item is stored, I can use simple math to figure out where the fifth item is stored. The reason that 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

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 (Links to an external site.)

GitHub - bgoonz/DATA_STRUC_PYTHON_NOTES

```
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
```

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 Simple Singly Linked List Node Class
2 value -> int
3 next -> LinkedListNode
```

```
1 class LinkedListNode:
2     def __init__(self, value):
3         self.value = value
4         self.next = None
5
6     def add_node(self, value):
7         # set current as a ref to self
8         current = self
9         # while there is still more nodes
10        while current.next is not None:
11            # traverse to the next node
12            current = current.next
13        # create a new node and set the ref from current.next to the new node
14        current.next = LinkedListNode(value)
15
16    def insert_node(self, value, target):
17        # create a new node with the value provided
18        new_node = LinkedListNode(value)
19        # set a ref to the current node
20        current = self
21        # while the current nodes value is not the target
22        while current.value != target:
23            # traverse to the next node
24            current = current.next
25        # set the new nodes next pointer to point toward the current nodes next pointer
26        new_node.next = current.next
27        # set the current nodes next to point to the new node
28        current.next = new_node
29
30
31    def print_ll(linked_list_node):
32        current = linked_list_node
33        while current is not None:
34            print(current.value)
35            current = current.next
36
37
38    def add_to_ll_storage(linked_list_node):
39        current = linked_list_node
40        while current is not None:
41            ll_storage.append(current)
42            current = current.next
43
44
45    ll_storage = []
46    L1 = LinkedListNode(34)
47    L1.next = LinkedListNode(45)
48    L1.next.next = LinkedListNode(90)
49    L1.add_node(12)
```

```
50 print_ll(L1)
51 L1.add_node(24)
52 print('-----\n')
53 print_ll(L1)
54 print('-----\n')
55 L1.add_node(102)
56 print_ll(L1)
57 L1.insert_node(123, 90)
58 print('-----\n')
59 print_ll(L1)
60 L1.insert_node(678, 34)
61 print('-----\n')
62 print_ll(L1)
63 L1.insert_node(999, 102)
64 print('-----\n')
65 print_ll(L1)
66
```

Result:



```
1 34
2 45
3 90
4 12
5 -----
6
7 34
8 45
9 90
10 12
11 24
12 -----
13
14 34
15 45
16 90
17 12
18 24
19 102
20 -----
21
22 34
23 45
24 90
25 123
26 12
27 24
28 102
29 -----
30
31 34
32 678
33 45
34 90
35 123
36 12
37 24
38 102
39 -----
40
41 34
42 678
43 45
44 90
45 123
46 12
47 24
48 102
49 999
```

```

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 pointer
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        - setup a current variable pointing to the head of the list
101        - set up a prev variable pointing to None
102        - set up a next variable pointing to None
103
104        - while the current ref is not none
105            - set next to the current.next
106            - set the current.next to prev
107            - set prev to current
108            - set current to next
109
110        - return prev
111
112        """
113        class LinkedListNode():
114            def __init__(self, value):
115                self.value = value
116                self.next = None
117
118            def reverse(head_of_list):
119                current = head_of_list
120                prev = None
121                next = None
122
123                while current:
124                    next = current.next
125                    current.next = prev
126                    prev = current
127                    current = next
128
129                return prev
130
131        class HashTableEntry:
132            """
133                Linked List hash table key/value pair
134            """
135            def __init__(self, key, value):
136                self.key = key
137                self.value = value
138                self.next = None
139
140        # Hash table can't have fewer than this many slots
141        MIN_CAPACITY = 8

```

```

142
143 [
144     0["Lou", 41] -> ["Bob", 41] -> None,
145     1["Steve", 41] -> None,
146     2["Jen", 41] -> None,
147     3["Dave", 41] -> None,
148     4None,
149     5["Hector", 34]-> None,
150     6["Lisa", 41] -> None,
151     7None,
152     8None,
153     9None
154 ]
155 class HashTable:
156     """
157         A hash table that with `capacity` buckets
158         that accepts string keys
159         Implement this.
160     """
161
162     def __init__(self, capacity):
163             self.capacity = capacity # Number of buckets in the hash table
164             self.storage = [None] * capacity
165             self.item_count = 0
166
167     def get_num_slots(self):
168         """
169             Return the length of the list you're using to hold the hash
170             table data. (Not the number of items stored in the hash table,
171             but the number of slots in the main list.)
172             One of the tests relies on this.
173             Implement this.
174         """
175         # Your code here
176
177     def get_load_factor(self):
178         """
179             Return the load factor for this hash table.
180             Implement this.
181         """
182         return len(self.storage) / self.capacity
183
184     def djb2(self, key):
185         """
186             DJB2 hash, 32-bit
187             Implement this, and/or FNV-1.
188         """
189         str_key = str(key).encode()
190
191         hash = FNV_offset_basis_64
192
193         for b in str_key:
194             hash *= FNV_prime_64
195             hash ^= b
196             hash &= 0xffffffffffffffff # 64-bit hash
197
198         return hash
199
200     def hash_index(self, key):
201         """
202             Take an arbitrary key and return a valid integer index
203             between within the storage capacity of the hash table.
204         """
205         return self.djb2(key) % self.capacity
206
207     def put(self, key, value):
208         """
209             Store the value with the given key.
210             Hash collisions should be handled with Linked List Chaining.
211             Implement this.
212         """
213         index = self.hash_index(key)
214
215         current_entry = self.storage[index]
216
217         while current_entry is not None and current_entry.key != key:

```

```
218         current_entry = current_entry.next
219
220     if current_entry is not None:
221         current_entry.value = value
222     else:
223         new_entry = HashTableEntry(key, value)
224         new_entry.next = self.storage[index]
225         self.storage[index] = new_entry
226
227     def delete(self, key):
228         """
229             Remove the value stored with the given key.
230             Print a warning if the key is not found.
231             Implement this.
232         """
233         # Your code here
234
235     def get(self, key):
236         """
237             Retrieve the value stored with the given key.
238             Returns None if the key is not found.
239             Implement this.
240         """
241         # Your code here
```

D3- Module 03 - Python III

Overview

A dictionary is like a list, but instead of accessing values with an index, you access values with a "key." A "key" can be any type of object (string, number, list, etc.). Also, unlike lists, dictionaries do not have an order.

Follow Along

Let's use a dictionary to create a collection that maps first names as keys (strings) to phone numbers as values.

```
1 phonebook = {} # creates an empty dictionary
2 phonebook["Abe"] = 4569874321
3 phonebook["Bill"] = 7659803241
4 phonebook["Barry"] = 6573214789
5
6 print(phonebook)
7 # {'Abe': 4569874321, 'Bill': 7659803241, 'Barry': 6573214789}
```

Instead of adding one key-value pair at a time, we can initialize the dictionary to have the same values.

```
1 phonebook = {
2     "Abe": 4569874321,
3     "Bill": 7659803241,
4     "Barry": 6573214789
5 }
6
7 print(phonebook)
8 # {'Abe': 4569874321, 'Bill': 7659803241, 'Barry': 6573214789}
```

We can iterate over a dictionary as we iterated over a list. We can use the `items()` method, which returns a tuple with the key and value for each item in the dictionary.

```
1 for name, number in phonebook.items():
2     print("Name: %s, Number: %s" % (name, number))
3
4 # Name: Abe, Number: 4569874321
5 # Name: Bill, Number: 7659803241
6 # Name: Barry, Number: 6573214789
```

To remove a key-value pair from a dictionary, you need to use the `del` keyword or use the `pop()` method available on dictionary objects. The difference is `pop()` deletes the item from the dictionary and returns the value. When you use the `del` keyword, you've written a statement that doesn't evaluate to anything.

```
1 phonebook = {
2     "Abe": 4569874321,
3     "Bill": 7659803241,
4     "Barry": 6573214789
5 }
6
7 del phonebook["Abe"]
8
```

```
9 print(phonebook.pop("Bill"))
10 # 7659803241
```



cs-unit-1-sprint-1-module-1-dictionaries-1

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-1-dictionaries-1#main.py>

- Home
- Grades
- Modules

Objective 01 - Perform basic dictionary operations

Overview

A dictionary is like a list, but instead of accessing values with an index, you access values with a "key." A "key" can be any type of object (string, number, list, etc.). Also, unlike lists, dictionaries do not have an order.

Follow Along

Let's use a dictionary to create a collection that maps first names as keys (strings) to phone numbers as values.

```
1 phonebook = {} # creates an empty dictionary
2 phonebook["Abe"] = 4569874321
3 phonebook["Bill"] = 7659803241
4 phonebook["Barry"] = 6573214789
5
6 print(phonebook)
7 # {'Abe': 4569874321, 'Bill': 7659803241, 'Barry': 6573214789}
```

Instead of adding one key-value pair at a time, we can initialize the dictionary to have the same values.

```
1 phonebook = {
2     "Abe": 4569874321,
3     "Bill": 7659803241,
4     "Barry": 6573214789
5 }
6
7 print(phonebook)
8 # {'Abe': 4569874321, 'Bill': 7659803241, 'Barry': 6573214789}
```

We can iterate over a dictionary as we iterated over a list. We can use the `items()` method, which returns a tuple with the key and value for each item in the dictionary.

```
1 for name, number in phonebook.items():
2     print("Name: %s, Number: %s" % (name, number))
```

```
3
4 # Name: Abe, Number: 4569874321
5 # Name: Bill, Number: 7659803241
6 # Name: Barry, Number: 6573214789
```

To remove a key-value pair from a dictionary, you need to use the `del` keyword or use the `pop()` method available on dictionary objects. The difference is `pop()` deletes the item from the dictionary and returns the value. When you use the `del` keyword, you've written a statement that doesn't evaluate to anything.

```
1 phonebook = {
2     "Abe": 4569874321,
3     "Bill": 7659803241,
4     "Barry": 6573214789
5 }
6
7 del phonebook["Abe"]
8
9 print(phonebook.pop("Bill"))
10 # 7659803241
```

Challenge

Additional Resources

- https://www.w3schools.com/python/python_dictionaries.asp (Links to an external site.)
- <https://docs.python.org/3/tutorial/datastructures.html#dictionaries> (Links to an external site.)



Objective 02 - Recognize mutable and immutable objects

Overview

In Python, everything is an object.

```
1 >>> a = 1
2 >>> b = "hello"
3 >>> c = [1,2,3]
4 >>> isinstance(a, object)
5 True
6 >>> isinstance(b, object)
7 True
8 >>> isinstance(c, object)
9 True
10 >>>
```

Additionally, all objects in Python have three things:

1. Identity
2. Type
3. Value

```
1 >>> a = 1
2 >>> # Identity
3 ... id(a)
4 4483164816
5 >>> # Type
6 ... type(a)
7 <class 'int'>
8 >>> # Value
9 ... a
10 1
11 >>>
```

Follow Along

Identity

An object's **identity** can never change once it has been created. You can think of an object's identity as its specific address in memory. In the code above, `a = 1` created a new object in memory whose identity is represented by the integer `4483164816`.

Python has an `is` operator that allows you to compare two object's identities.

```
1 >>> a = 1
2 >>> b = 2
3 >>> a is b
4 False
5 >>> b = a
6 >>> a is b
7 True
8 >>>
```

In the code above, we first assign `1` to the variable `a`. Then, we assign `2` to the variable `b`. These are two different objects in memory and thus have different identities. We verify that they are different by using the `is` operator, which returns `False`. The line `b = a` assigns the variable `b` the object that the variable `a` is pointed to. Now, both `a` and `b` are referencing the same object in memory. We can use the `id()` function to verify that this is the case as well:

```
1 >>> id(a)
2 4483164816
3 >>> id(b)
4 4483164816
5 >>>
```

Type

The **type** of an object determines what are its possible values and what operations that object supports. The `type()` function will return what type an object is:

```
1 >>> a = 'Hello'
2 >>> type(a)
3 <class 'str'>
4 >>> b = 100
5 >>> type(b)
6 <class 'int'>
7 >>> c = True
8 >>> type(c)
9 <class 'bool'>
10 >>>
```

Just like an object's identity, once an object is created, its identity can never change. It's an object's type that determines whether an object is **mutable** or **immutable**.

Value

The value of some objects *can be changed* after they are created. The value of some objects *cannot be changed* after they are created. If the object's value can be changed, that object is considered to be **mutable** (changeable). If the object's value cannot be changed, that object is considered to be **immutable** (unchangeable).

Mutable Objects

A mutable object is an object whose value can be changed after it is created. The word **mutable** is defined as:

liable to change

The following types of objects are mutable:

- list
- set
- dict
- byte array
- instances of user-defined classes

Let's look at a few examples in code:

Lists

```
1 >>> my_list = ['laughter', 'happiness', 'love']
2 >>> type(my_list)
3 <class 'list'>
4 >>> my_list[2] = 'joy'
5 >>> my_list.append('excellent')
6 >>> my_list
7 ['laughter', 'happiness', 'joy', 'excellent']
8 >>>
```

In the first line, we create a list object with three elements and assign it to the variable `my_list`. Then, because lists are mutable, we change `'love'` at index 2 to be `'joy'` instead. We also can grow our list by appending a new element to the list.

Sets

```
1 >>> my_set = {'laughter', 'happiness', 'love'}
2 >>> type(my_set)
3 <class 'set'>
4 >>> my_set.add('happy')
5 >>> my_set
6 {'love', 'happy', 'happiness', 'laughter'}
7 >>> my_set.remove('happiness')
8 >>> my_set
9 {'love', 'happy', 'laughter'}
```

In the first line, we create a set object with three elements and assign it to the variable `my_set`. Because set objects are mutable, we can add `'happy'` to the set and remove `'happiness'` from the set.

Dicts

```

1 >>> my_dict = {"first_name": "Mattieu", "last_name": "Ricard"}
2 >>> type(my_dict)
3 <class 'dict'>
4 >>> my_dict["location"] = "Nepal"
5 >>> my_dict
6 {'first_name': 'Mattieu', 'last_name': 'Ricard', 'location': 'Nepal'}
7 >>> del my_dict['location']
8 >>> my_dict
9 {'first_name': 'Mattieu', 'last_name': 'Ricard'}

```

On line one, we create a dict object that has two key-value pairs. Then, because dict objects are mutable, we add key-value pair "location": "Nepal". Last, we delete that same key-value pair.

Mutable objects work great when you know you will likely need to change the size of the object as you use and interact with it. Changing mutable objects is cheap (because you don't have to copy all existing elements to a new object).

Aliasing with Mutable Objects

Below, I'm going to walk through what happens when you **alias** a mutable object. In Python, aliasing happens whenever a variable's value is assigned to another variable because variables are just names that store references to values.

Let me illustrate this with a helpful code visualizer tool called [Python Tutor](#) (Links to an external site.):

Python 3.6
(known limitations)

```

1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)

```

Edit this code

line that just executed
next line to execute

<< First < Prev Next >> Last >>

Step 3 of 4

https://tk-assets.lambdaschool.com/ba46ee2f-6bb4-421e-8be7-cba3a55eedcf_Untitled.png

On line 1, we instantiate a new list object with three elements (1, 2, and 3). The name `my_list_orig` is the variable that we assign the new list to.

Python 3.6
(known limitations)

```

1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)

```

Edit this code

line that just executed
next line to execute

<< First < Prev Next >> Last >>

Step 4 of 4

https://tk-assets.lambdaschool.com/23cd8845-e086-4cf6-9b50-70b37a11731b_Untitled-2.png

Then, on line 2, we create an alias of `my_list_orig` by pointing `my_list_alias` to whatever object `my_list_orig` is pointing at. Notice in the image above that there is still only one list object. However, there are two variables in the global frame, and they are both

pointing to the same object.

Python 3.6
(known limitations)

```
1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)
```

[Edit this code](#)

line that just executed
next line to execute

<< First < Prev Next > >>

Step 4 of 4

https://tk-assets.lambdaschool.com/604c130d-254c-4126-87a8-49625e676ef4_Untitled-3.png

Frames Objects

Global frame

my_list_orig

my_list_alias

list

0	1	2	3	4
1	2	3	4	

On line 3, we append a new element to `my_list_orig`. Notice that, because both variables are referencing the same object, even though we appended to `my_list_orig`, we also modified `my_list_alias`.

Python 3.6
(known limitations)

```
1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)
```

[Edit this code](#)

line that just executed
next line to execute

<< First < Prev Next > >>

Done running (4 steps)

https://tk-assets.lambdaschool.com/f1655834-f68c-4b49-95ca-93d4a1578423_Untitled-4.png

Frames Objects

Global frame

my_list_orig

my_list_alias

list

0	1	2	4
2	3	4	

On line 4, we removed the element `1` from `my_list_orig`. Notice, just like when we added to the list, `my_list_alias` is also affected.

This behavior is something you need to be aware of if you ever use aliasing with mutable objects in your code.

Immutable Objects

An immutable object is an object whose value cannot be changed after it is created. Immutable means *not changeable*. Anytime you try to update the value of an immutable object, a new object is created instead.

The following types are immutable:

- Numbers (int, float, complex)
- Strings
- Bytes
- Booleans
- Tuples

Immutable objects are useful when you want to make sure that the object you created will always maintain the same value. Immutable objects are more *expensive* to change (in terms of time and space complexity) because changing the object requires making a copy of the existing object.

Let's look at a few examples:

Numbers

```
1 >>> my_int = 1
2 >>> id(my_int)
3 4513307280
4 >>> type(my_int)
5 <class 'int'>
6 >>> my_int
7 1
8 >>> my_int = 2
9 >>> id(my_int)
10 4513307312
11 >>> type(my_int)
12 <class 'int'>
13 >>> my_int
14 2
15 >>>
```

In the code above, the first line creates a new int object, and the variable `my_int` now points at that object. You can see that this object has `int` for its type, `4513307280` for its identity (location in memory), and `1` for its value.

Then, we assign `2` to `my_int` which creates a whole new object and assigns it to the variable `my_int`. This object has `int` for its type, `4513307312` for its identity (which you can see is different from the first object), and `2` for its value.

Strings

Let's look at how string concatenation works in Python. Remember that str objects are immutable.

```
1 >>> my_str = 'a'
2 >>> type(my_str)
3 <class 'str'>
4 >>> id(my_str)
5 140716674193840
6 >>> my_str
7 'a'
8 >>> my_str += 'b'
9 >>> type(my_str)
10 <class 'str'>
11 >>> id(my_str)
12 140716674658992
13 >>> my_str
14 'ab'
15 >>>
```

So, on line 1, we create a string object with the value `'a'` and assign it to the variable `my_str`. We verify that the object is of type `str`, we print its identity (`140716674193840`) and print its value.

Then, we concatenate `'b'` onto the existing string with the line `my_str += 'b'`. Now, because string objects are immutable, we cannot change a string object's value after it has been created. To concatenate, we create a new string object and assign the value `'ab'` to that object.

This behavior in Python is vital to be aware of when working with string concatenation. If you have to add and remove frequently from a string, this will be inefficient if you work with string objects directly.

Tuples

Tuples are an immutable container of names, where each name has an unchangeable (immutable) binding to an object in memory. You cannot change the bindings of the names to the objects.

```
1 >>> my_tuple = ('love', [1,2,3], True)
2 >>> my_tuple[0]
3 'love'
4 >>> my_tuple[0] = 'laughter'
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: 'tuple' object does not support item assignment
8 >>>
```

Here we created a tuple using `()` to denote the tuple literal syntax. Just like a list, tuples can contain elements of any type.

Above, we've included a string, a list, and a boolean as our tuple elements. We are proving the tuple object's immutability by showing the error that occurs when trying to assign a new item to a position in the tuple.

One thing that often causes confusion surrounding the immutability of tuples in Python is demonstrated by the following behavior:

```
1 >>> my_tuple[1] = [4,5,6]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: 'tuple' object does not support item assignment
5 >>> id(my_tuple[1])
6 140716674620864
7 >>> my_tuple[1][0] = 4
8 >>> my_tuple[1][1] = 5
9 >>> my_tuple[1][2] = 6
10 >>> my_tuple[1]
11 [4, 5, 6]
12 >>> my_tuple
13 ('love', [4, 5, 6], True)
14 >>> id(my_tuple[1])
15 140716674620864
16 >>>
```

Notice that we cannot create a new list object and bind it to the name at position 1 of our tuple. This is demonstrated when

`my_tuple[1] = [4,5,6]` raises a `TypeError`. However, we can assign new objects to the list that is at position 1 of our tuple? Why is that? Well, what do we know about lists in Python? Lists are mutable objects. So, we can modify a list without creating a new object. So, when we are modifying the list directly (instead of assigning a new object), it doesn't affect our tuple's immutability. Notice that the identity (`140716674620864`) of the list at `my_tuple[1]` doesn't change after replacing its three elements with `4`, `5`, and `6`.

Passing Objects to Functions

Mutable and immutable objects are not treated the same when they are passed as arguments to functions. When mutable objects are passed into a function, they are passed by reference. So, suppose you change the mutable object that was passed in as an argument. In that case, you are changing the original object as well.

Mutable Objects as Arguments

```
1 >>> my_list = [1,2,3]
2 >>> def append_num_to_list(lst, num):
3 ...     lst.append(num)
4 ...
5 >>> append_num_to_list(my_list, 4)
6 >>> my_list
7 [1, 2, 3, 4]
8 >>>
```

```

Python 3.6
(known limitations)

1 my_list = [1,2,3]
2
3
4 def append_num_to_list(lst, num):
5     lst.append(num)
6
7
8 append_num_to_list(my_list, 4)

Edit this code

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>
Step 1 of 6

```

<https://tk-assets.lambdaschool.com/5528e90f-2784-4199-b520-a4d03adccbbc Mutable-object-passed-as-argument-to-function.gif>

Notice that when `append_num_to_list` is called and `my_list` is passed in as an argument. When `my_list` is bound to `lst` in that stack frame, `lst` points to the original `my_list` in memory. The function call did not create a copy of `my_list`. This behavior is because lists are mutable objects in Python.

Immutable Objects as Arguments

Next, let's see how Python behaves when we pass an immutable object as an argument to a function:

```

1 >>> my_string = "I am an immutable object."
2 >>> def concatenate_string_to_string(orig_string, string_to_add):
3 ...     return orig_string + string_to_add
4 ...
5 >>> concatenate_string_to_string(my_string, " I hope!")
6 'I am an immutable object. I hope!'
7 >>> my_string
8 'I am an immutable object.'
9 >>>

```

```

Python 3.6
(known limitations)

1 my_string = "I am an immutable object."
2
3
4 def concatenate_string_to_string(orig_string, string_to_
5     return orig_string + string_to_add
6
7
8 concatenate_string_to_string(my_string, " I hope!")

Edit this code

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>
Step 1 of 6

```

<https://tk-assets.lambdaschool.com/3e6a1461-9853-4494-8c17-33919e641eb0 Immutable-object-passed-argument-to-function.gif>

Notice when an immutable object is passed into a function, the object is copied and bound to the parameter name. In the example above, when `my_string` is passed into `concatenate_string_to_string`, `my_string` is copied to a new object bound to the name `orig_string`.

Challenge



- - Home
 - Grades
 - Modules

Objective 02 - Recognize mutable and immutable objects

Overview

In Python, everything is an object.

```
1  >>> a = 1
2  >>> b = "hello"
3  >>> c = [1,2,3]
4  >>> isinstance(a, object)
5  True
6  >>> isinstance(b, object)
7  True
8  >>> isinstance(c, object)
9  True
10 >>>
```

Additionally, all objects in Python have three things:

1. Identity
2. Type
3. Value

```
1  >>> a = 1
2  >>> # Identity
3  ... id(a)
4  4483164816
5  >>> # Type
6  ... type(a)
7  <class 'int'>
8  >>> # Value
9  ... a
10 1
11 >>>
```

Follow Along

Identity

An object's **identity** can never change once it has been created. You can think of an object's identity as its specific address in memory. In the code above, `a = 1` created a new object in memory whose identity is represented by the integer `4483164816`.

Python has an `is` operator that allows you to compare two object's identities.

```
1 >>> a = 1
2 >>> b = 2
3 >>> a is b
4 False
5 >>> b = a
6 >>> a is b
7 True
8 >>>
```

In the code above, we first assign `1` to the variable `a`. Then, we assign `2` to the variable `b`. These are two different objects in memory and thus have different identities. We verify that they are different by using the `is` operator, which returns `False`. The line `b = a` assigns the variable `b` the object that the variable `a` is pointed to. Now, both `a` and `b` are referencing the same object in memory. We can use the `id()` function to verify that this is the case as well:

```
1 >>> id(a)
2 4483164816
3 >>> id(b)
4 4483164816
5 >>>
```

Type

The **type** of an object determines what are its possible values and what operations that object supports. The `type()` function will return what type an object is:

```
1 >>> a = 'Hello'
2 >>> type(a)
3 <class 'str'>
4 >>> b = 100
5 >>> type(b)
6 <class 'int'>
7 >>> c = True
8 >>> type(c)
9 <class 'bool'>
10 >>>
```

Just like an object's identity, once an object is created, its identity can never change. It's an object's type that determines whether an object is **mutable** or **immutable**.

Value

The value of some objects *can be changed* after they are created. The value of some objects *cannot be changed* after they are created. If the object's value can be changed, that object is considered to be **mutable** (changeable). If the object's value cannot be changed, that object is considered to be **immutable** (unchangeable).

Mutable Objects

A mutable object is an object whose value can be changed after it is created. The word **mutable** is defined as:

liable to change

The following types of objects are mutable:

- list
- set
- dict
- byte array
- instances of user-defined classes

Let's look at a few examples in code:

Lists

```
1 >>> my_list = ['laughter', 'happiness', 'love']
2 >>> type(my_list)
3 <class 'list'>
4 >>> my_list[2] = 'joy'
5 >>> my_list.append('excellent')
6 >>> my_list
7 ['laughter', 'happiness', 'joy', 'excellent']
8 >>>
```

In the first line, we create a list object with three elements and assign it to the variable `my_list`. Then, because lists are mutable, we change `'love'` at index 2 to be `'joy'` instead. We also can grow our list by appending a new element to the list.

Sets

```
1 >>> my_set = {'laughter', 'happiness', 'love'}
2 >>> type(my_set)
3 <class 'set'>
4 >>> my_set.add('happy')
5 >>> my_set
6 {'love', 'happy', 'happiness', 'laughter'}
7 >>> my_set.remove('happiness')
8 >>> my_set
9 {'love', 'happy', 'laughter'}
```

In the first line, we create a set object with three elements and assign it to the variable `my_set`. Because set objects are mutable, we can add `'happy'` to the set and remove `'happiness'` from the set.

Dicts

```
1 >>> my_dict = {"first_name": "Mattieu", "last_name": "Ricard"}
2 >>> type(my_dict)
3 <class 'dict'>
4 >>> my_dict["location"] = "Nepal"
5 >>> my_dict
6 {'first_name': 'Mattieu', 'last_name': 'Ricard', 'location': 'Nepal'}
7 >>> del my_dict['location']
8 >>> my_dict
9 {'first_name': 'Mattieu', 'last_name': 'Ricard'}
```

On line one, we create a dict object that has two key-value pairs. Then, because dict objects are mutable, we add key-value pair `"location": "Nepal"`. Last, we delete that same key-value pair.

Mutable objects work great when you know you will likely need to change the size of the object as you use and interact with it. Changing mutable objects is cheap (because you don't have to copy all existing elements to a new object).

Aliasing with Mutable Objects

Below, I'm going to walk through what happens when you **alias** a mutable object. In Python, aliasing happens whenever a variable's value is assigned to another variable because variables are just names that store references to values.

Let me illustrate this with a helpful code visualizer tool called [Python Tutor](#) (Links to an external site.):

Python 3.6
(known limitations)

```
1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)
```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > >>

Step 3 of 4

https://tk-assets.lambdaschool.com/ba46ee2f-6bb4-421e-8be7-cba3a55eedcf_Untitled.png

On line 1, we instantiate a new list object with three elements (1 , 2 , and 3). The name `my_list_orig` is the variable that we assign the new list to.

Python 3.6
(known limitations)

```
1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)
```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > >>

Step 4 of 4

https://tk-assets.lambdaschool.com/23cd8845-e086-4cf6-9b50-70b37a11731b_Untitled-2.png

Then, on line 2, we create an alias of `my_list_orig` by pointing `my_list_alias` to whatever object `my_list_orig` is pointing at. Notice in the image above that there is still only one list object. However, there are two variables in the global frame, and they are both pointing to the same object.

Python 3.6
(known limitations)

```
1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
4 my_list_orig.remove(1)
```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > >>

Step 4 of 4

https://tk-assets.lambdaschool.com/604c130d-254c-4126-87a8-49625e676ef4_Untitled-3.png

On line 3, we append a new element to `my_list_orig`. Notice that, because both variables are referencing the same object, even though we appended to `my_list_orig`, we also modified `my_list_alias`.

The screenshot shows a Python 3.6 debugger interface. On the left, code is shown with line 4 highlighted:

```

Python 3.6
(known limitations)

1 my_list_orig = [1,2,3]
2 my_list_alias = my_list_orig
3 my_list_orig.append(4)
→ 4 my_list_orig.remove(1)

```

Below the code are status indicators: a green arrow for the line just executed and a red arrow for the next line to execute. At the bottom are navigation buttons: << First, < Prev, Next >, and Last >>. A message "Done running (4 steps)" is displayed.

On the right, the "Frames" and "Objects" panes show the state of the Global frame. The "list" object contains the values 0, 2, 3, and 4. Both `my_list_orig` and `my_list_alias` point to the same list object.

https://tk-assets.lambdaschool.com/f1655834-f68c-4b49-95ca-93d4a1578423_Untitled-4.png

On line 4, we removed the element `1` from `my_list_orig`. Notice, just like when we added to the list, `my_list_alias` is also affected.

This behavior is something you need to be aware of if you ever use aliasing with mutable objects in your code.

Immutable Objects

An immutable object is an object whose value cannot be changed after it is created. Immutable means *not changeable*. Anytime you try to update the value of an immutable object, a new object is created instead.

The following types are immutable:

- Numbers (`int`, `float`, `complex`)
- Strings
- Bytes
- Booleans
- Tuples

Immutable objects are useful when you want to make sure that the object you created will always maintain the same value. Immutable objects are more *expensive* to change (in terms of time and space complexity) because changing the object requires making a copy of the existing object.

Let's look at a few examples:

Numbers

```

1 >>> my_int = 1
2 >>> id(my_int)
3 4513307280
4 >>> type(my_int)
5 <class 'int'>
6 >>> my_int
7 1
8 >>> my_int = 2
9 >>> id(my_int)
10 4513307312
11 >>> type(my_int)
12 <class 'int'>
13 >>> my_int
14 2
15 >>>

```

In the code above, the first line creates a new int object, and the variable `my_int` now points at that object. You can see that this object has `int` for its type, `4513307280` for its identity (location in memory), and `1` for its value.

Then, we assign `2` to `my_int` which creates a whole new object and assigns it to the variable `my_int`. This object has `int` for its type, `4513307312` for its identity (which you can see is different from the first object), and `2` for its value.

Strings

Let's look at how string concatenation works in Python. Remember that str objects are immutable.

```
1 >>> my_str = 'a'
2 >>> type(my_str)
3 <class 'str'>
4 >>> id(my_str)
5 140716674193840
6 >>> my_str
7 'a'
8 >>> my_str += 'b'
9 >>> type(my_str)
10 <class 'str'>
11 >>> id(my_str)
12 140716674658992
13 >>> my_str
14 'ab'
15 >>>
```

So, on line 1, we create a string object with the value `'a'` and assign it to the variable `my_str`. We verify that the object is of type `str`, we print its identity (`140716674193840`) and print its value.

Then, we concatenate `'b'` onto the existing string with the line `my_str += 'b'`. Now, because string objects are immutable, we cannot change a string object's value after it has been created. To concatenate, we create a new string object and assign the value `'ab'` to that object.

This behavior in Python is vital to be aware of when working with string concatenation. If you have to add and remove frequently from a string, this will be inefficient if you work with string objects directly.

Tuples

Tuples are an immutable container of names, where each name has an unchangeable (immutable) binding to an object in memory. You cannot change the bindings of the names to the objects.

```
1 >>> my_tuple = ('love', [1,2,3], True)
2 >>> my_tuple[0]
3 'love'
4 >>> my_tuple[0] = 'laughter'
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: 'tuple' object does not support item assignment
8 >>>
```

Here we created a tuple using `(` and `)` to denote the tuple literal syntax. Just like a list, tuples can contain elements of any type. Above, we've included a string, a list, and a boolean as our tuple elements. We are proving the tuple object's immutability by showing the error that occurs when trying to assign a new item to a position in the tuple.

One thing that often causes confusion surrounding the immutability of tuples in Python is demonstrated by the following behavior:

```

1 >>> my_tuple[1] = [4,5,6]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: 'tuple' object does not support item assignment
5 >>> id(my_tuple[1])
6 140716674620864
7 >>> my_tuple[1][0] = 4
8 >>> my_tuple[1][1] = 5
9 >>> my_tuple[1][2] = 6
10 >>> my_tuple[1]
11 [4, 5, 6]
12 >>> my_tuple
13 ('love', [4, 5, 6], True)
14 >>> id(my_tuple[1])
15 140716674620864
16 >>>

```

Notice that we cannot create a new list object and bind it to the name at position 1 of our tuple. This is demonstrated when `my_tuple[1] = [4,5,6]` raises a `TypeError`. However, we can assign new objects to the list that is at position 1 of our tuple? Why is that? Well, what do we know about lists in Python? Lists are mutable objects. So, we can modify a list without creating a new object. So, when we are modifying the list directly (instead of assigning a new object), it doesn't affect our tuple's immutability. Notice that the identity (`140716674620864`) of the list at `my_tuple[1]` doesn't change after replacing its three elements with `4`, `5`, and `6`.

Passing Objects to Functions

Mutable and immutable objects are not treated the same when they are passed as arguments to functions. When mutable objects are passed into a function, they are passed by reference. So, suppose you change the mutable object that was passed in as an argument. In that case, you are changing the original object as well.

Mutable Objects as Arguments

```

1 >>> my_list = [1,2,3]
2 >>> def append_num_to_list(lst, num):
3 ...     lst.append(num)
4 ...
5 >>> append_num_to_list(my_list, 4)
6 >>> my_list
7 [1, 2, 3, 4]
8 >>>

```

Python 3.6
(known limitations)

```

→ 1 my_list = [1,2,3]
2
3
4 def append_num_to_list(lst, num):
5     lst.append(num)
6
7
8 append_num_to_list(my_list, 4)

```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > > Last >>

Step 1 of 6

<https://tk-assets.lambdaschool.com/5528e90f-2784-4199-b520-a4d03adccbcb Mutable-object-passed-as-argument-to-function.gif>

Notice that when `append_num_to_list` is called and `my_list` is passed in as an argument. When `my_list` is bound to `lst` in that stack frame, `lst` points to the original `my_list` in memory. The function call did not create a copy of `my_list`. This behavior is

because lists are mutable objects in Python.

Immutable Objects as Arguments

Next, let's see how Python behaves when we pass an immutable object as an argument to a function:

```
1 >>> my_string = "I am an immutable object."
2 >>> def concatenate_string_to_string(orig_string, string_to_add):
3     ... return orig_string + string_to_add
4 ...
5 >>> concatenate_string_to_string(my_string, " I hope!")
6 'I am an immutable object. I hope!'
7 >>> my_string
8 'I am an immutable object.'
9 >>>
```

The screenshot shows a Python 3.6 interactive shell interface. The code area contains the following Python code:

```
1 my_string = "I am an immutable object."
2
3 def concatenate_string_to_string(orig_string, string_to_add):
4     ... return orig_string + string_to_add
5
6 concatenate_string_to_string(my_string, " I hope!")
```

Below the code, there are status indicators: "line that just executed" (grey arrow) and "next line to execute" (red arrow). At the bottom, there are navigation buttons: '<< First', '< Prev', 'Next' (which has a mouse cursor over it), and 'Last >>'. A progress bar at the bottom indicates "Step 1 of 6".

https://tk-assets.lambdaschool.com/3e6a1461-9853-4494-8c17-33919e641eb0_immutable-object-passed-argument-to-function.gif

Notice when an immutable object is passed into a function, the object is copied and bound to the parameter name. In the example above, when `my_string` is passed into `concatenate_string_to_string`, `my_string` is copied to a new object bound to the name `orig_string`.

Challenge

Additional Resources

- [Mutable vs. Immutable Objects in Python - A Visual and Hands-On Guide](#) (Links to an external site.)
- [Python Basics: Mutable vs. Immutable Objects](#) (Links to an external site.)
- [What are mutable and immutable objects in Python3?](#) (Links to an external site.)



cs-unit-1-sprint-1-module-3-mutable-and-immutable-objects-1

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-3-mutable-and-immutable-objects-1#main.py>



Objective 03 - Compare the time complexity of different approaches to a problem using Big O notation

Overview

What is an algorithm?

An algorithm is a set of instructions for accomplishing a task. Within this broad definition, we could call every piece of code an algorithm.

How do we measure how "good" an algorithm is?

After coming up with a first-pass solution to a problem, we need to measure how "good" our answer is. Will it stand up to the test of millions of users? Is it fast enough that our users will be blown away by how quickly they get their results? Or will tortuously slow speeds cause lag that scares them all away?

When given a choice between different algorithms, we want to choose the most efficient algorithm (considering both *time* and *space* efficiency depending on our needs).

Note: It is common for your first solution to work with a few items or users and break as you add more. Making sure that the solutions scale is something all developers must look out for.

What is Big O notation?

We need a way to talk about efficiency (number of operations in the worst case) in a more general sense.

Big O notation is the language we use for describing how efficient an algorithm is.

The specific terms of Big O notation describe how fast the runtime grows (relative to the input size), focusing on when the input gets extremely large.

Why do we focus on the growth of runtime versus exact runtime? The actual runtime depends on the specific computer running the algorithm, so we cannot compare efficiencies that way. By focusing on the general growth, we can avoid exact runtime differences between machines and environments.

We also talk about runtime relative to the input size because we need to express our speed in terms of *something*. So we show the speed of the algorithm in terms of the input size. That way, we can see how the speed reacts as the input size grows.

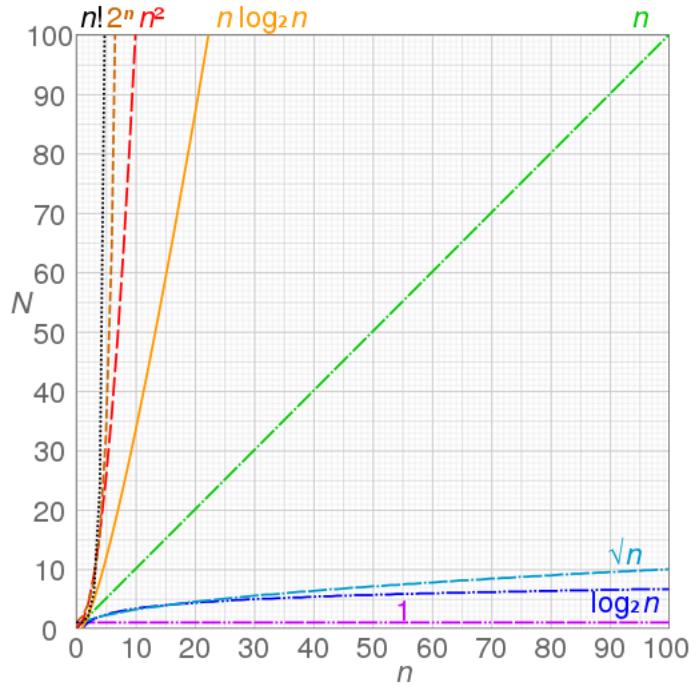
We don't care about speed when the input size is small. The differences in speed are likely to be minimal when the input size is small. When the input size gets enormous, we can see the differences in efficiency between algorithms.

Common Big O run times

Refer to the table below to see a list of the most common runtimes. The table is ordered from fastest to slowest.

Classification	Description
Constant $O(1)$	The runtime is entirely unaffected by the input size. This is the ideal solution.
Logarithmic $O(\log n)$	As the input size increases, the runtime will grow slightly slower. This is a pretty good solution.
Linear $O(n)$	As the input size increases, the runtime will grow at the same rate. This is a pretty good solution.
Polynomial $O(n^c)$	As the input size increases, the runtime will grow at a faster rate. This might work for small inputs but is not a scalable solution.
Exponential $O(c^n)$	As the input size increases, the runtime will grow at a much faster rate. This solution is inefficient.
Factorial $O(n!)$	As the input size increases, the runtime will grow astronomically, even with relatively small inputs. This solution is exceptionally inefficient.

Besides the table, it's also essential to look at the curves of these different runtimes.



https://tk-assets.lambdaschool.com/1b27038a-098f-46e5-bc20-03be9a3480b9_68747470733a2f2f746b2d6173736574732e6c616d6264617363686f6f6c2e636f6d2f65343335376235662d316436332d343463642d623861302d3439353732363061653965635f556e7469746c6564312e706e67.png

Again, `n` represents the size of the data, and on the chart above, `N` represents the number of operations. This visualization should help illustrate why `O(1)` or `O(log n)` is the most desirable.

Note: Big O only matters for large data sets. An `O(n^3)` solution is adequate, as long as you can guarantee that your datasets will always be small.

A few examples

Let's look at a few different examples of Python functions that print something to the output. For each of these, the input will be `items`.

Constant Time $O(1)$

```
1 def print_only_one_thing(list_of_things):
2     print(list_of_things[0])
```

Why is this constant time? Because no matter how large or small the input is (1,000,000 or 10), the number of computations within the function is the same.

Linear Time $O(n)$

```
1 def print_list(list_of_things):
2     for thing in list_of_things:
3         print(thing)
```

Why is this classified as linear time? Because the speed of the algorithm increases at the same rate as the input size. If `list_of_things` has ten items, then the function will print ten times. If it has 10,000 items, then the function will print 10,000 times.

Quadratic Time $O(n^2)$

```
1 def print_permutations(list_of_things):
2     for thing_one in list_of_things:
3         for thing_two in list_of_things:
4             print(thing_one, thing_two)
```

Why is this quadratic time? The clue is the nested for loops. These nested for loops mean that for each item in `list_of_things` (the outer loop), we iterate through every item in `list_of_things` (the inner loop). For an input size of `n`, we have to print `n * n` times or n^2 times.

What are we supposed to do with the constants?

What if we had a function like this?

```
1 def do_a_bunch_of_stuff(list_of_things): # O(1 + n/2 + 2000)
2     last_idx = len(list_of_things) - 1
3     print(list_of_things[last_idx]) # O(1)
4
5     middle_idx = len(list_of_things) / 2
6     idx = 0
7     while idx < middle_idx: # O(n/2)
8         print(list_of_things[idx])
9         idx = idx + 1
10
11    for num in range(2000): # O(2000)
12        print(num)
```

`print(items[last_idx])` is constant time because it doesn't change as the input changes. So, that portion of the function is `O(1)`.

The while loop that prints up to the middle index is $1/2$ of whatever the input size is; we can say that portion of the function is `O(n/2)`.

The final portion will run 2000 times, no matter the size of the input.

So, putting it all together, we could say that the efficiency is `O(1 + n/2 + 2000)`. However, we don't say this. We describe this function as having linear time `O(n)` because we drop all of the constants. Why do we cut all of the constants? Because as the input size gets huge, adding 2000 or dividing by 2 has minimal effect on the algorithm's performance.

Most significant term

Let's consider the following function:

```
1 def do_different_things_in_the_same_function(list_of_things): # O(n + n^2)
2     # print all each item in the list
3     for thing in list_of_things: # O(n)
4         print(thing)
5
6     # print every possible pair of things in the list
7     for thing_one in list_of_things: # O(n * n) = O(n^2)
8         for thing_two in list_of_things:
9             print(thing_one, thing_two)
```

We could describe this function as `O(n + n2)`; however, we only need to keep the essential term, `n2`, so this would be `O(n2)`. Why can we do this? Because as the input size (`n`) gets larger and larger, the less significant terms have less effect, and only the most significant term is important.

Big O represents the worst-case

Let's consider the following function:

```
1 def find_thing(list_of_things, thing_we_are_trying_to_find):
2     for thing in list_of_things:
3         if thing == thing_we_are_trying_to_find:
4             return True
5
6 return False
```

What would the result be if it just so happens that the `thing_we_are_trying_to_find` in `list_of_things` is the very first item in the list? The function would only have to look at one item in `list_of_things` before returning. In this case, it would be $O(1)$. But, when we talk about a function's complexity, we usually assume the "worst case." What would the "worst-case" be? It would be if it were the last item in `list_of_things`. In that case, we would have to look through all the `list_of_things`, and that complexity would be $O(n)$.

Note: When talking about runtime complexity in casual conversation, engineers often blur the distinction between big theta and big O notation. In reality, these are two distinct ways of describing an algorithm. Big theta gives both an upper and a lower bound for the running time. Big O only provides an upper bound. Refer to the following articles for a deeper dive: [Big-Theta notation \(Links to an external site.\)](#) and [Big-O notation \(Links to an external site.\)](#).

Do constants ever matter?

Complexity analysis with Big O notation is a valuable tool. It would be best if you got in the habit of thinking about the efficiency of the algorithms you write and use in your code. However, just because two algorithms have the same Big O notation doesn't mean they are equal.

Imagine you have a script that takes 1 hour to run. By improving the function, you can divide that runtime by six, and now it only takes 10 minutes to run. With Big O notation, $O(n)$ and $O(n/6)$ can both be written as $O(n)$, but that doesn't mean it isn't worth optimizing the script to save 50 minutes every time the script runs.

That being said, there is a term you should become familiar with: [premature optimization \(xkcd: Optimization \(Links to an external site.\)\)](#). Sometimes, you can sacrifice readability or spend too much time on something to improve its efficiency. Depending on the situation, it could be that having a finished product to iterate on is more important than maximally efficient code. It is your job as a developer to know when making your code more efficient is necessary. You will always be making calculated tradeoffs between runtime, memory, development time, readability, and maintainability. It takes time to develop the wisdom to strike the right balance depending on the scenario.

Follow Along

Let's look at a few code snippets and classify their runtime complexity using Big O notation.

```
1 def foo(n):
2     i = 1
3     while i < n:
4         print(i)
5         i *= 2
```

First, let's think about what the above function is doing. It's printing `i` ...but `i` is not being incremented by 1, as we usually see. It's *doubled* every time we run the loop. So, for example, if `n = 100`, then the final result would be...

```
1 1
2 2
3 4
```

```
4 8  
5 16  
6 32  
7 64
```

Or if `n = 10`, then we would print...

```
1 1  
2 2  
3 4  
4 8
```

We can use the process of elimination to narrow down which runtime classification makes sense for this algorithm. The number of times the loop runs seems to vary based on the value of `n`, so this is NOT $O(1)$.

From the above examples, we can also see that the number of times the loop runs is increasing *slower* than the input size is increasing. `n` must be *doubled* before the loop will run one more time. We can eliminate classifications such as $O(n \log n)$, $O(n^c)$, $O(c^n)$, and $O(n!)$.

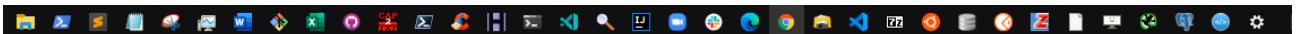
The only two options left at this point are logarithmic and linear. Since the two growth rates (input, the number of operations) are not the same, **this function must run in logarithmic time!**

Challenge



[cs-unit-1-sprint-1-module-2-time-complexity-1](https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-2-time-complexity-1)

<https://replit.com/@bgoonz/cs-unit-1-sprint-1-module-2-time-complexity-1>



D4-Module 04 - Python IV

Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and basic operations of a static array

Overview

Python does not have a static array data type. However, lists are built on dynamic arrays. As you will see, dynamic arrays rely on an underlying static array to work. So while you won't be creating and using this data structure directly, it is still essential to understand.

A data structure is a *structure* that is designed for holding information in a particular way. A static array is a data structure that is designed for storing information sequentially (in order). For example, if you were to store the English alphabet in a static array, you would expect the "B" character to right next to both the "A" character and the "C" character. Additionally, every position within the static array is labeled with an index. So, if you wanted to access the first item in the static array, you would expect that item to have an index of 0. The second item would have an index of 1. The third item would have an index of 2. This pattern continues for the entire capacity of the static array.

Follow Along

Time and Space Complexity

Lookup

To look up an item by index in an array is constant time ($O(1)$). If you have the specific index of an object in an array, the computations to find that item in memory are all constant time as well.

Append

Adding an item to an array is constant time ($O(1)$). We always have a reference point to the last thing in a static array, so we can insert an item after the current end.

Insert

Unless you are inserting an item at the end of the list, items must be shifted over to make room for the new information you add to the static array. It's like if you had a chain of people stretched out, holding hands, in a line. The first person in the line is butted right up against a wall, and there is no room on one side of him. If someone wanted to join the end of the line, the people already in the line wouldn't have to do anything ($O(1)$). However, if you wanted to join the beginning of the line, every single person would have to move over (away from the wall) ($O(n)$) so that you would have room to join. If you wanted to join the line somewhere in the middle, only the people to your one side would have to shift to make room for you. In the computer, this shifting is moving information from one address in memory to another. Each move takes time.

Delete

Just like insertions, deletions are only efficient ($O(1)$) when they are done at the end of the static array. If something is deleted from any other position in the array, the items have to be moved over, so there isn't any empty space left. Remember, static arrays can be a good data structure because retrieving information from a specific index is fast. It is fast because we can ensure that information is consistently stored in sequence right next to each other. That way, we can always be confident that whatever information is at index 5 is the sixth item in the array. If we left empty spaces in the middle of our static array, we would no longer ensure that this was true.

Space

The space complexity of an array is linear ($O(n)$). Each item in the array will take up space in memory.

Strengths

Static arrays are great to use when you need a data structure to retrieve information from a specific index efficiently. This is because, as we explained earlier, accessing any specific index in a static array involves a simple mathematical computation (starting index + (size of each item * index)). This computation is done in $O(1)$ time and is not affected by the static array size at all. If you need a data structure where you are likely only to append items (add them to the end of the list), a static array also works great. When you add a new item to the end of the list, nothing has to be shifted over or moved in memory, so that operation is very efficient ($O(1)$).

Weaknesses

There are situations where static arrays are not the best data structure to use for storing your information. What about if you don't know how much information you need to store? Or if the amount of information you need to store is likely to fluctuate or change frequently. In this case, a static array is not good. The reason is that when you create a static array, you have to know and declare the size of that array. That way, your computer can separate off a chunk of memory that is the exact right size for storing that static array. If you run out of room in the static array, you can't simply make it bigger; you have to create a brand new, bigger static array. You have to copy each item from the first static array into the newer, bigger one.

Another reason that static arrays are not always the best choice to use for storing information is that they are inefficient unless you are performing operations at the end of the static array. They are inefficient because if you want to insert or delete something at the beginning (or the middle of the list), all the items to the right of that index must be moved over. If you delete something, everything has to be shifted over, so there isn't an empty index in the middle of your data. If you insert something, all the items have to shift over to make room for the new item before inserting it.

What about array slicing?

You often encounter a scenario where you want to use a subset of items from an existing array. Array slicing is when you take a subset from an existing array and allocate a new array with just the items from the slice.

In Python, the syntax looks like this:

```
my_list[start_index:end_index]
```

The default start index is 0, and if you leave off the end_index, the slice will capture through the end of the list.

```
1 my_list[:] # This would be all of the items in my_list
2 my_list[:5] # This would be the items from index 0 to 4
3 my_list[5:] # This would be the items from index 5 to the end of the list
```

You might be wondering, what is the time and space complexity of slicing an array? To understand the complexity, you need to know what is happening behind the scenes when you take a slice of an array. First, you are *actually allocating a new list*. Second, you copy all of the items in your slice from the original array into the newly allocated list. This means that you have an $O(n)$ time cost (for the copying) and an $O(n)$ space cost for the newly allocated list.

You must keep these facts in mind and account for them when using a slice in your code. It's not a free operation.

Challenge

1. Draw out what happens to a static array when you insert an item at the beginning of the array.
2. Draw out what happens to a static array when you delete an item from the array's beginning.

Additional Resources

- <https://www.hackerearth.com/practice/data-structures/arrays/1-d/tutorial/> (Links to an external site.)
- <https://www.pythongcentral.io/how-to-slice-listsarrays-and-tuples-in-python/>



Objective 02 - Describe the differences between in-place and out-of-place algorithms

Overview

In-Place

An in-place function modifies or destroys the state of the input data when it is run. For instance, if you write a function that squares every integer in an input list, an in-place version of this function would change the data in the list that was passed in. It would not create a new list and return the new list. In-place functions are more space-efficient because they don't create new variables directly tied to the input size. However, to get that space-efficiency, you have to risk that the function's user may end up changing state to the input accidentally.

Imagine a scenario where you have an antique map that you are using to navigate on a hike. You end up needing directions, and when you come across another hiker, you ask them for help. The person helping you has two options. They can take your antique map, use a pen, and mark it up with their notations that will help you navigate. However, you most likely didn't want those annotations to be on your map forever. The other option would be to find another piece of paper and have the person helping you write out their annotations on that. This way, your original antique map doesn't have to be modified. However, now you have two maps that you have to carry around on your hike.

Out-of-Place

In contrast to in-place functions, out-of-place functions don't modify or destroy the input state when they are run. Any changes done to the input are done to a copy of the input, not the original that was passed in. This is why they are less space-efficient. If you have a list of 1,000,000 items that you want to square, you first have to make a copy of that list. Now, you have two lists of 1,000,000 items. However, you avoid any side-effects that might be unintended.

Pass By Reference or Value

In Python, some function arguments are passed in by their actual value, and some are passed in as a reference to the object in memory. Primitive values like integers, floats, and strings are passed in by their actual value. So, if you call a function and pass in the integer `2` when you reference that value by the named parameter of the function, you can't change `2` in memory. However, non-primitive objects like lists or dictionaries are passed in as references to that object in memory. So, if you call a function and pass in the dictionary `{"name": "Matt"}` when you reference that dictionary using the named parameter, you are changing the original object that was passed in. For objects that are passed in by reference, they must be copied to a new variable before they are modified if you want to avoid side effects.

When should I use an in-place function or algorithm?

It would be best if you always defaulted to using an out-of-place function. This is a safer default to avoid unintended side-effects in your program. However, there are scenarios that you might encounter where you need to be extremely space-efficient. In that case, you might have to use an in-place function to work within the particular space-constraints you've been given.

Follow Along

Here is an example of a function that triples each number in an input list. This function does this *in-place*:

```
1 def append_exclamations(str_list):
2     for idx, item in enumerate(str_list):
3         str_list[idx] += "!"
```

Now, since this is an in-place function, watch what happens when we use it:

```
1 >>> my_list = ["Matt", "Beej", "Sean"]
2 >>> append_exclamations(my_list)
3 >>> my_list
4 ['Matt!', 'Beej!', 'Sean!']
```

`my_list` was modified when I called the function, and the function only returned the default return value of `None`.

Let's now write a similar function, but this time we will do it *out-of-place*:

```
1 def append_exclamations(str_list):
2     # Create a new empty list that has the same length as the input list
3     loud_list = [None] * len(str_list)
4     for idx, item in enumerate(str_list):
5         # insert the modified string into the new list
6         loud_list[idx] = item + "!"
7     # Since we didn't modify the input list, we need to return the new list to
8     # the function caller
9     return loud_list
```

Look what happens when we use this function:

```
1 >>> my_list = ["Matt", "Beej", "Sean"]
2 >>> my_new_louder_list = append_exclamations(my_list)
3 >>> my_list
4 ['Matt', 'Beej', 'Sean']
5 >>> my_new_louder_list
6 ['Matt!', 'Beej!', 'Sean!']
7 >>>
```

Notice how we had to store the returned list in a new variable. Also, notice that it didn't modify the list that we passed in when we called the function.

Challenge

1. In your own words, describe the difference between an in-place algorithm and an out-of-place algorithm.
2. In your own words, explain when it is an excellent choice to use an in-place algorithm.

Additional Resources

- <https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/>



Objective 03 - Recall the time and space complexity, the strengths and weaknesses, and basic operations of a dynamic array

Overview

Remember how we said you had to know how much information you were going to store when you created a static array? Well, with a dynamic array, you don't have to know. You don't have to declare a size when you instantiate a dynamic array. That makes it better in scenarios where the amount of information you need to store is unknown or is likely to fluctuate.

Time and Space Complexity

Lookup

To look up an item by index in an array is constant time ($O(1)$). If you have the specific index of an item in an array, the computations to find that item in memory are all constant time as well.

Append

Adding an item to an array is constant time ($O(1)$) in the average case. However, in the worst case, the cost is $O(n)$ (this will be explained in more detail below).

Insert

In the worst case, inserting an item is linear time ($O(n)$). When you insert into an array, all the items – starting at the index we are inserting into – have to be shifted one index. These items have to be "moved over" to make room for the new item being inserted. The worst-case scenario is inserting at the 0th index, and every item in the array has to shift over.

Delete

In the worst case, deleting an item is linear time ($O(n)$). For any item you delete (unless it is the last item), all of the items after that index have to be shifted over to fill the now blank spot in the array. Remember, arrays store data in sequential order, so if we delete an item, we cannot just leave that space blank. If we left the space blank, it would ruin the quick lookup time. To have a fast lookup time, we need to be able to rely on the distance from the start of the array to whatever index we are trying to access.

Space

The space complexity of an array is linear ($O(n)$). Each item in the array will take up space in memory.

Strengths

Again, probably the dynamic array's biggest strength is not having to know or worry about the size of the data structure. It can grow to accommodate your data as needed. And, you don't have to manage this growth; the data structure itself grows when necessary. Dynamic arrays also have some of the same strengths as a static array. They also have efficient lookups ($O(1)$) when you have a specific index that you want to retrieve from.

Weaknesses

The main weakness of the dynamic array is related to its strength. To not have to worry about or manage the array's size, when the array runs out of room, it has to grow to accommodate more items. So, let's say your dynamic array is currently set up to store ten items. If it's full and you try to add an 11th item, the data structure can't just assume that there is a spot available right after the 10th item. It actually creates a new, bigger array and then copies all of the first ten items into the new array, and finally, it adds the 11th item. We will talk a bit

more about how this works below. Additionally, dynamic arrays have the same weaknesses as static arrays, slow insertions and deletions ($O(n)$).

Follow Along

Doubling Appends

Underneath the hood of a dynamic array is a static array. When you create a dynamic array, it is a static array that keeps track of the starting index, the index of the last item that it stores, and the index for the last slot in its capacity. This brings up an important point. An array has a size and a capacity. An array's size is how many items it is storing at the moment. Its capacity is how many items it could store before it runs out of room.

So, let's say that your dynamic array instantiates with an underlying static array with a capacity of 10 and a size of 0 when you create it. Then, you add ten items to the array. Now, it has a capacity of 10 and a size of 10. If you now go to append an 11th item to the array, you've run out of capacity. Here is where the *dynamic* of the dynamic array comes into play. The data structure will create a new underlying static array with a capacity twice the size of the original underlying static array. It will then copy the ten original items into the new array and finally add the 11th item. The cost of copying the original items into the new array is $O(n)$. So, when we say that, in the worst-case, an append on a dynamic array has a time-complexity of $O(n)$, this is why. However, all the other appends still have a time-complexity of $O(1)$. So, in the average case append, the time-complexity is still efficient. Also, consider that as the array's capacity keeps doubling, the doublings will occur less and less frequently.

Challenge

1. What type in Python is a dynamic array?
2. In your own words, explain why the worst-case time cost of appending to a dynamic array is $O(n)$.
3. What is the difference between the size of a dynamic array and the capacity of a dynamic array?

Additional Resources

- <https://www.youtube.com/watch?v=qTb1sZX74K0> (Links to an external site.)



Static Arrays vs. Dynamic Arrays

<https://www.youtube.com/watch?v=qTb1sZX74K0>



Array and String Manipulation

This module project requires you to answer some multiple-choice questions related to the module's objectives. Additionally, you must continue developing your problem-solving skills by completing coding challenges related to its content.



wk18

→ [wk17](#)

/cirriculum/untitled-3

→ [D1-Module 01 - Python I](#)

/cirriculum/untitled-3/untitled-2

→ [D3- Module 03 - Python III](#)

/cirriculum/untitled-3/untitled-1

→ [D4-Module 04 - Python IV](#)

/cirriculum/untitled-3/untitled

→ [wk18](#)

/cirriculum/untitled-2

Overview

My Notion Notes:



Notion – The all-in-one workspace for your notes, tasks, wikis, and databases.

<https://www.notion.so/webdevhub42/D3-Aux-Resources-82d1b5c6ff8f4e139902e7c678f555fe#344c1246d74949dbb8469221eb17358d>

Overview

In this sprint, you will learn about number bases, character encoding, and how values are stored in memory. We'll also talk all about hash tables, a very powerful data structure for getting more performance and flexibility out of your algorithms. Finally, we'll look at optimizing searching through data, as well as the powerful programming technique of recursion.

Number Bases and Character Encoding

This module will teach about RAM, number bases, fixed-width integers, how arrays are stored in memory, and character encoding. These basics allow you to better understand how other data structures are stored in memory and have an intuitive sense of the time complexity of specific data structures' operations.

Hash Tables I

In this module, you will learn about the properties of hash tables, hash functions, and how to implement a hash table in Python. Hash tables are arguably one of the most important and common data structures you use and encounter to solve algorithmic coding challenges.

Hash Tables II

In this module, you will learn about hash collisions and implement a complete hash table in Python that takes collisions into account.

Searching and Recursion

This module will teach about logarithms, linear search, binary search, and recursion. This material sets the stage for learning about traversal techniques of trees and graphs. Additionally, using a binary search is a common optimization technique that may come up during a technical interview.



iframes - CodeSandbox

<https://codesandbox.io/s/iframes-v9xfe?file=/index.html>

D1- Module 01 - Number Bases and Character Encoding

Objective 01 - Understand random access memory (RAM) as it relates to data structures

Overview

Your computer has something called random access memory (RAM). Sometimes, people say "memory" when referring to RAM.

Follow Along

One thing that might come to your mind is that there are different types of memory on your computer. What about storing things like videos, text documents, and applications? Are those in "memory"? There is a distinction between "storage" and "memory". Things like videos and files are stored on a disc, not in RAM. RAM is faster than disc storage, but there isn't as much space available. Disc storage has more space, but it is slower.

Think of RAM like a set of numbered, sequential mailboxes. Just like a set of mailboxes with numbered addresses, RAM is also sequential and has numbered addresses.

Now, just like you can put something in a mailbox, you can also put something in RAM. Things that you put in RAM, we can call variables. Each "box" in RAM has an **address**.

Each one of the "boxes" (memory addresses) in our set of mailboxes (RAM) holds 8 bits. You can think of each bit like a tiny switch that can either be "on" or "off." "On" is represented by a `1`, and "off" is represented by a `0`.

Bits are often thought about in groups. A group of 8 bits is called a byte. Each "box" in RAM can hold 1 byte (8 bits).

Now, a computer has more than just disc storage and RAM inside of it. There is also a processor. And, in between the processor and the RAM is something called a memory controller. The memory controller can access each box in RAM directly. It is as if the memory controller had tubes connected to each box of the set of mailboxes. Through those tubes, the memory controller can send and receive information directly to each box in RAM.

Why is the direct connection between the memory controller and each box in RAM meaningful? It's so that the memory controller can jump around to which box it wants to communicate with quickly. Even though the boxes are in sequential order, the memory controller doesn't have to go through the boxes in order. It can access the first one, then jump to one somewhere in the middle, and then access one at the end. Because there is a direct connection, this is done quickly.

Whenever you use a computer, you are very concerned with the speed of the computer you are using. So, computer designers made a way to optimize for speed when accessing items in RAM. Whenever a processor accesses a box in RAM, it also accesses and stores the boxes near it. Often, if you are accessing one thing in RAM, it's likely that the next thing you need to access is nearby. That's why keeping a copy of nearby items in the cache speeds things up.

Whenever the processor reads something (say, the player's position in an old adventure game) out of RAM, it adds it to the cache to use it again in the future. Then, when it needs something else from RAM, it will go to the cache for it. As you can see, the cache helps the processor by saving execution cycles required to go out and read something from RAM.

The processor, not RAM, has the actual cache. The memory controller keeps track of what goes into and comes out of the cache.

We can think of it in several ways. Perhaps, the processor can use the cache as a temporary area to keep a copy of its last actions just in case it needs to reread them.

There is one caveat – it is not as if "everything" goes out to RAM and then gets inserted into the cache. In reality, the cache holds only a handful of memory addresses from RAM. Also, note that these few memory addresses in the cache can be accessed faster than other storage locations.

Challenge

Draw a model of how a processor interacts with the cache, memory controller, and RAM whenever it needs to read or write from memory.

Additional Resources

- https://en.wikipedia.org/wiki/Random-access_memory (Links to an external site.)
- https://en.wikipedia.org/wiki/Memory_controller (Links to an external site.)
- https://en.wikipedia.org/wiki/CPU_cache (Links to an external site.)



Objective 02 - Convert back and forth from decimal to binary

Overview

Computers use the binary number system, so we will represent all of our variables in the binary number system.

Instead of 10 digits like 1, 2, 3, 4, 5, 6, 7, 8, 9, and 0, the binary number system only has two possible digits, 1 and 0. Another way to think of it is that computers only have switches (bits) that can be in an "off state" or an "on state."

Follow Along

Before we try to understand the binary number system, let's review how the decimal number system works. Let's look at the number "1001" in decimal.

Even though there are two "1" digits in this number, they don't represent the same quantity. The leftmost "1" represents one thousand, and the rightmost "1" represents one unit. The "0"s in-between represent the tens place and the hundreds place.

So this "1001" in base ten represents "1 thousand, 0 hundreds, 0 tens, and 1 one."

Each successive digit in the base 10 number system is a power of ten. The ones place is $10^0 = 1$. The tens place is $10^1 = 10$. The hundreds place is $10^2 = 100$. This pattern continues on and on.

This pattern holds for other number systems as well. In the binary system, each successive digit represents a different power of 2. The first digit represents $2^0 = 1$. The second digit represents $2^1 = 2$. The third digit represents $2^2 = 4$. Again, this pattern continues on and on.

So, what if the number "1001" was in binary and not decimal? What would it represent then? Well, if we read it right to left, we have a "1" in the ones place, a "0" in the twos place, a "0" in the fours place, and a "1" in the eights place. We add these values up ($8 + 0 + 0 + 1$) which equals 9.

Below, is a table that shows how to count up to 8 in binary and decimal:

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

Challenge

Convert the following decimal numbers into binary numbers:

1. 25
2. 63
3. 9
4. 111

Additional Resources

- <https://www.mathsisfun.com/binary-number-system.html> (Links to an external site.)
- <https://www.mathsisfun.com/definitions/decimal-number-system.html>



Objective 03 - Understand how fixed-width integers are stored in memory

Overview

We now know that things are stored in RAM using binary, and each "box" in RAM holds 1 byte (8 bits). What does that mean for what we can store in RAM? Let's say we have 1 byte of RAM to use. How many different numbers can we represent using only this 1 byte?

Remember that each digit in a binary number is a successive power of 2. If we have 8 bits to use, we can store $2^8 = 256$ different numbers in 1 byte.

Follow Along

Let's see if we can find a pattern:

- With one bit, we can express two numbers (0 and 1)
 - With two bits, for each of the first numbers (0 or 1), we can put a 0 or a 1 after it, so we can express four numbers
 - With three bits, we can express eight numbers.

Every time we add a new bit, we double the number of possible numbers we can express in binary. This pattern can be generalized as 2^n and $2^8 = 256$.

Often, computers use 4 bytes (32 bits) to represent our variables, meaning that we can express as many as 4 billion (2^{32}) possible values. Similarly, computers may use 8 bytes (64 bits) to represent our variables and can express over 10 billion (2^{64}).

The 2^8 X in the binary number system is called the **bitsize**. Eight bytes of memory are called "8-bit", and 16 bytes are called "16-bit" etc.

In theory, you could use less space to represent smaller integers. For instance, in binary, the number one is represented by `1`. So, technically, to store one in binary, you only need one bit. But computers don't usually do this. Many integers take a fixed amount of space, no matter what number they might have in them. So, even though you only need one bit to represent the number one, the computer would still use 32 or 64 bits to do so.

So, if a variable represents a fixed-width integer, it doesn't matter if it has the value `0` or `123,456`; the amount of space it takes up in RAM is the same.

The computer can store numbers like 3, 60000000, or -14 in 32 bits, one of the "fixed-width integers" we discussed earlier. All of these fixed-width integers take up constant space ($O(1)$).

Storing numbers as fixed-width integers introduces a trade-off. We have constant space complexity, and because each integer has a constant and expected number of bits, simple mathematical operations only take constant time. The cost of having an integer as fixed-width is that there is a limit to the number of integers you can represent.

Challenge

1. What is the number of possible integer values you can store with 4 bytes? How did you make that calculation?
 2. What is the number of possible integer values you can store with 8 bytes? How did you make that calculation?

Additional Resources

- <https://vladris.com/blog/2018/10/13/arithmetic-overflow-and-underflow.html> (Links to an external site.)



Objective 04 - Describe, in general terms, how arrays are stored in memory and the time complexity of lookups

Overview

When writing programs, you likely need to store several numbers, not just one integer.

Follow Along

So, let's say we wanted to write a program that allowed us to keep track of the number of hours we spent studying that day. We will round the number of hours to the nearest whole number to store them as fixed-width integers. Additionally, each day's hours will be represented by eight bits in binary.

So, we will start at memory address 0 in RAM, and each day, store the number of hours we studied in that "box" of RAM. For our first day that we are tracking, we store an 8-bit binary integer in "box" number 0. On the second day, we store an 8-bit binary integer in "box" number 1. This pattern continues.

Now, I'm sure you've already used an array when you are programming. An array is just an ordered sequential collection of data. Well, RAM is already structured like this. Right? Our days where we track the number of hours that we are studying are in sequential order in RAM.

Knowing this information, what can we do if we want to look up how many hours we studied on day 5 (index 4 because of zero-indexing)? Because all of the information is stored in sequential order, we can do simple math. If you are looking for the day 5 information (index 4), you need to know what the starting item address is 0 and then add 4 (the index). Or, if the starting address was 5 and you were looking for the 10th index, you'd go to memory address 15 ($5 + 10$).

This math works because we are using one "box" in memory for each day's record. If we were using 64-bit integers that take up 8 "boxes" in RAM, we would have to slightly adjust our math. In this case, we would have to multiply the index we were looking for by the number of bytes each record was stored in. So, if we were storing 64-bit integers (8 bytes) and wanted to find the item with index 4, and the starting index was 0, we would go to memory address $0 + (4 * 8) = 32$.

Because accessing information from a specific index involves this simple mathematical computation, accessing items in an array is a constant time operation. For the mathematical computations to be consistent and straightforward, arrays have to follow specific rules. Each item in the array has to take up the same number of bytes in RAM. Also, each item has to be stored right next to the previous item in RAM. If there are any gaps or interruptions in the array, then the simple mathematical computation for accessing a particular item no longer works.

Challenge

Let's say you need to store an array of 64-bit integers. Your array needs to have enough capacity for 24 integers. How many 1-byte slots of memory need to be allocated to store this array?

Additional Resources

- https://en.wikipedia.org/wiki/Array_data_type



Objective 05 - Describe character encoding and how strings are stored in memory

Overview

In this example, we will store some strings. A string, as we know, is just a bunch of characters or letters. One straightforward way to store a string is an array, so let's see how we can define some mappings to make it easier to store strings in arrays.

Follow Along

To use our 8-bit slots in memory, we need a way to encode each character in a string in 8-bits. One common character encoding to do this is called "ASCII". Here's how the alphabet is encoded in ASCII:

Letter	Encoding
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000
I	01001001
J	01001010
K	01001011
L	01001100
M	01001101
N	01001110
O	01001111
P	01010000
Q	01010001
R	01010010
S	01010011
T	01010100
U	01010101
V	01010110
W	01010111
X	01011000
Y	01011001
Z	01011010

Since we can express characters as 8-bit integers, we can express strings as arrays of 8-bit characters.

For example, we could represent LAMBDA like so:

1	L	->	01001100
2	A	->	01000001
3	M	->	01001101
4	B	->	01000010
5	D	->	01000100
6	A	->	01000001

Each character, once it was encoded, could be stored as one 8-bit slot in memory.

Challenge

Draw out a model of a section of memory that stores the string "Computer Science" as an array of 8-bit ASCII characters.

Additional Resources

- https://www.w3schools.com/charsets/ref_html_ascii.asp



Arithmetic Overflow and Underflow

Arithmetic Overflow and Underflow

Arithmetic overflow happens when an arithmetic operation results in a value that is outside the range of values representable by the expression's type. For example, the following C++ code prints 0:

```
1 uint16_t x = 65535;  
2 x++;  
3  
4 std::cout << x;
```

x is an unsigned 16 bit integer, which can represent values between 0 and 65535. If x is 65535 and we increment it, the value becomes 65536 but that value cannot be represented by a uint16_t. This is an overflow. In this case, C++ wraps the value around and x becomes 0.

Similarly, an underflow occurs when an arithmetic operation generates a result that is below the smallest representable value of the expression's type:

```
1 uint16_t x = 0;  
2 x--;  
3  
4 std::cout << x;
```

The above prints 65535, as the result of decrementing 0 is -1, which cannot be represented by an uint16_t.

Before we digging into overflow behavior, we need to understand how computers represent numbers.

Number Representations

Arbitrarily large integers

Python provides support for arbitrarily large integers: unlike C++, where the bit width (number of bits used to represent a number) is fixed, we can have integers of any size:

```
print(10**100)
```

Prints

Why don't all languages provide such support? The answer is performance. The underlying hardware the code runs on uses fixed-width integers, so performing arithmetic on fixed-width integer types becomes a single CPU instruction. On the other hand, supporting arbitrarily large integers usually involves writing code to determine how many bits a given value or the result of an arithmetic operation needs and convert that into an array of fixed-width integers large enough to hold that value. The added overhead of this is non-trivial, so unlike Python, most other mainstream languages offer only fixed-width integers and support arbitrarily large integers only explicitly, via libraries.

Unsigned integers

Unsigned integers are represented as a sequence of N bits, thus being able to represent numbers between 0 and $2^N - 1$. An unsigned 8-bit integer can store any value between 0 and 255, an unsigned 16-bit integer can store any value between 0 and 65535, an unsigned 32-bit integer between 0 and 4294967295, and an unsigned 64-bit integer between 0 and 18446744073709551615.

Unsigned integer representation is trivial.

Signed integers

Signed integers are usually represented in two's complement.

Positive numbers are encoded the same as unsigned binary numbers described above. Negative numbers are encoded as two's complement of their absolute value. For example, an 8-bit representation of -3 is 283.

The most significant bit is always 1 for negative numbers and 0 for positive numbers or 0.

With this representation, N bits can encode a signed integer between $2^{N-1} - 1$ and -2^N . So 8 bits can encode an integer between -128 and 127.

Handling Overflow

If the result of an arithmetic operation cannot fit the type, there are several approaches we can take and different programming languages employ different strategies. These are:

- Exceptions
- Wrap-around
- Saturation

All of these approaches have their pros and cons.

Exceptions

The safest approach is to treat an arithmetic overflow as an exception. This usually gets rid of security vulnerabilities and treats any overflow as an exceptional scenario. In this case an exception is thrown (or an error returned) whenever an arithmetic operation overflows or underflows.

This is usually desirable from a security/safety perspective, but the trade-off is in performance: the downside of this approach is that all arithmetic operations need to be checked for overflow (underlying hardware usually does not do this natively) and exceptions need to be handled by callers.

Wrap-around

The default behavior in C++, wrap-around simply continues from the smallest possible value in case of overflow or from the largest possible value in case of underflow. For unsigned integers, this is equivalent to modulo arithmetic. For example, for an `int8_t`, which can represent values between -128 and 127, wrap-around would make $127 + 1$ be -128 and similarly $-128 - 1$ be 127.

This is usually the most efficient way to perform arithmetic as no checking is involved. Most hardware uses wrap-around as it can simply discard overflowing bits to achieve the result. The two's complement representation of 127 is 01111111. The two's complement representation of 128 is 10000000. With this representation, adding 1 to 127 naturally makes it 128.

This is also the most unsafe implementation as it can lead to unexpected behavior and exploitable security holes[\[1\]](#).

Saturation

Saturation means clamping the value within the allowed range, so on overflow, we would simply stop at the largest representable value. On underflow, we would stop at the smallest representable value. In our 8-bit signed integer example, we would now have $127 + 1$ be 127 and $-128 - 1$ be -128. There are several advantages with this approach: for one, the resulting values on overflow and underflow are the

closest to the “real” values we would get if operating without constraints. A lot of physical systems naturally lend themselves to saturation. Imagine, for example, a thermostat which can only operate within a range of temperature.

The downsides of this approach are results which might be surprising and the fact that properties of arithmetic operations like associativity no longer hold: $(120 + 10) + (-10)$ is 117, but $120 + (10 + (-10))$ is 120.

Detecting Overflow and Underflow

Let’s now see how we can tell whether an arithmetic operation overflow while operating only within the range of values representable by a given type.

For a type which can represent any value between some MIN and MAX, we observe that an addition overflow means $a + b > \text{MAX}$, while an underflow means $a + b < \text{MIN}$ (note a and b can be negative, so adding them could produce a value that would be under our minimum representable value).

We can detect overflow and underflow by checking, if $b \geq 0$, that $a > \text{MAX} - b$, otherwise with $b < 0$, that $a < \text{MIN} - b$.

The reason this works is that, if b is greater than or equal to 0, we can safely subtract it from MAX (if it were negative, subtracting it would cause an overflow). So with this in mind, we are simply saying that $a + b > \text{MAX}$ is equivalent to $a > \text{MAX} - b$ (subtracting b on both sides). We also observe that $a + b$ can never underflow if b is greater than or equal to 0 because, regardless how small a is, adding a positive number to it will make it larger not smaller.

If b is less than 0, then by the same logic we cannot possibly overflow - regardless how large a is, adding b to it would make it smaller. In this case we only need to check for underflow. Here we observe that subtracting a negative number from MIN is safe - it will increase MIN. So by subtracting b on both sides of $a + b < \text{MIN}$, we get $a < \text{MIN} - b$.

The following code implements these two checks:

```
1 #include <limits>
2
3 template <typename T>
4 constexpr bool AdditionOverflows(const T& a, const T& b) {
5     return (b >= 0) && (a > std::numeric_limits<T>::max() - b);
6 }
7
8 template <typename T>
9 constexpr bool AdditionUnderflows(const T& a, const T& b) {
10    return (b < 0) && (a < std::numeric_limits<T>::min() - b);
11 }
```

Detecting overflow or underflow for subtraction is very similar, as subtracting b from a is the equivalent of adding $-b$ to a , thus we only need to adjust the checks. $a - b > \text{MAX}$ means $a > \text{MAX} + b$ if b is negative (so we don’t cause an overflow during the check), while $a - b < \text{MIN}$ means $a < \text{MIN} + b$ if b is greater than or equal to 0:

```
1 template <typename T>
2 constexpr bool SubtractionOverflows(const T& a, const T& b) {
3     return (b < 0) && (a > std::numeric_limits<T>::max() + b);
4 }
5
6 template <typename T>
7 constexpr bool SubtractionUnderflows(const T& a, const T& b) {
8     return (b >= 0) && (a < std::numeric_limits<T>::min() + b);
9 }
```

Detecting overflow for multiplication is more interesting. $a * b > \text{MAX}$ can happen if $b \geq 0$, $a \geq 0$, and $a > \text{MAX} / b$ or when $b < 0$, $a < 0$, and $a < \text{MAX} / b$ (dividing $a * b > \text{MAX}$ on both sides by b , a negative number, flips the sign of the inequality).

Underflow can happen only when one of the numbers is negative and the other one isn't. So if $b \geq 0$, $a < 0$, and $a < \text{MIN} / b$ or if $b < 0$, $a \geq 0$, and $a > \text{MIN} / b$.

We can implement the checks as follows:

```
1 template <typename T>
2 constexpr bool MultiplicationOverflows(const T& a, const T& b) {
3     return ((b >= 0) && (a >= 0) && (a > std::numeric_limits<T>::max() / b))
4         || ((b < 0) && (a < 0) && (a < std::numeric_limits<T>::max() / b));
5 }
6
7 template <typename T>
8 constexpr bool MultiplicationUnderflows(const T& a, const T& b) {
9     return ((b >= 0) && (a < 0) && (a < std::numeric_limits<T>::min() / b))
10        || ((b < 0) && (a >= 0) && (a > std::numeric_limits<T>::min() / b));
11 }
```

Note integer division cannot possibly underflow. The single overflow that can happen is due to the fact that in two's complement representation, we can represent one more negative number than positives, as 0 is, in a sense, positive with this representation (the sign bit is not set for 0). An 8-bit signed integer can represent 128 positive values (0 to 127) and 128 negative values (-1 to -128). Overflow can only happen when we change the sign of the smallest possible value we can represent by dividing it with -1. -128 / -1 becomes 128, which is an overflow. This is the only case we need to check for:

```
1 template <typename T>
2 constexpr bool DivisionOverflows(const T& a, const T& b) {
3     return (a == std::numeric_limits<T>::min()) && (b == -1)
4         && (a != 0);
5 }
```

Note that unsigned integers can never overflow, so once we confirm that a is the smallest possible value and b is -1, we also check to ensure a is not 0.

We are explicitly not looking at division by 0, which is part of the same safe arithmetic topic. This post focuses on overflow and underflow only.

Handling Overflow and Underflow

Now that we can detect overflows and underflows, we can implement a couple of policies to handle them. Wrap-around is the default behavior in C++, so let's look at the other two possibilities. We will implement a couple of types templated on an integer type T , with overflow and underflow handlers:

```
1 template <typename T>
2 struct Policy {
3     static constexpr T OnOverflow() { /* ... */ }
4     static constexpr T OnUnderflow() { /* ... */ }
5 };
```

The throwing policy looks like this:

```
1 struct ArithmeticException : std::exception {};
2 struct ArithmeticOverflowException : ArithmeticException {};
3 struct ArithmeticUnderflowException : ArithmeticException {};
4
5 template <typename T>
6 struct ThrowingPolicy {
```

```

7     static constexpr T OnOverflow() {
8         throw new ArithmeticOverflowException{};
9     }
10
11    static constexpr T OnUnderflow() {
12        throw new ArithmeticUnderflowException{};
13    }
14 };

```

The saturation policy is:

```

1 template <typename T>
2 struct SaturationPolicy {
3     static constexpr T OnOverflow() {
4         return std::numeric_limits<T>::max();
5     }
6
7     static constexpr T OnUnderflow() {
8         return std::numeric_limits<T>::min();
9     }
10 };

```

Safe Arithmetic

Now that we have all the required pieces, we can create a type that wraps an integer type and implements all the arithmetic operations checking for overflow or underflow. The type is templated on a policy for handling overflows and underflows:

```

1 template <typename T, template<typename> typename Policy>
2 struct Integer
3 {
4     T value;
5
6     constexpr Integer<T, Policy> operator+(
7         const Integer<T, Policy>& other) const {
8         if (AdditionOverflows(value, other.value))
9             return { Policy<T>::OnOverflow() };
10
11         if (AdditionUnderflows(value, other.value))
12             return { Policy<T>::OnUnderflow() };
13
14         return { value + other.value };
15     }
16
17     constexpr Integer<T, Policy> operator-(
18         const Integer<T, Policy>& other) const {
19         if (SubtractionOverflows(value, other.value))
20             return { Policy<T>::OnOverflow() };
21
22         if (SubtractionUnderflows(value, other.value))
23             return { Policy<T>::OnUnderflow() };
24
25         return { value - other.value };
26     }
27
28     constexpr Integer<T, Policy> operator*(
29         const Integer<T, Policy>& other) const {
30         if (MultiplicationOverflows(value, other.value))
31             return { Policy<T>::OnOverflow() };
32
33         if (MultiplicationUnderflows(value, other.value))
34             return { Policy<T>::OnUnderflow() };
35
36         return { value * other.value };
37     }
38
39     constexpr Integer<T, Policy> operator/(
40         const Integer<T, Policy>& other) const {

```

```

41         if (DivisionOverflows(value, other.value))
42             return { Policy<T>::OnOverflow(); }
43
44         return { value / other.value };
45     }
46 };

```

Now we can wrap an integer type with this and perform safe arithmetic:

```

1 Integer<int8_t, ThrowingPolicy> a{ 64 };
2 Integer<int8_t, ThrowingPolicy> b{ 2 };
3
4 // Throws
5 Integer<int8_t, ThrowingPolicy> result = a * b;

```

This is a simple implementation for illustrative purposes. The Integer type currently only defines addition, subtraction, multiplication, and division. A complete implementation would handle multiple other operators, like pre and post increment, implicit casting from T etc.

The generic overflow and underflow checks can be specialized for unsigned types so that we don't redundantly check for $b < 0$ for a type which cannot represent negative numbers. Similarly, we wouldn't worry, for example, about addition underflowing for an unsigned type.

We can also extend our safe arithmetic to not only rely on the standard numeric_limits, but also allow users to clamp values between user-defined minimum and maximum values.

For a production-ready safe arithmetic library, I recommend you check out David LeBlanc's [SafeInt](#).

Summary

This post covered arithmetic overflow and underflow, and ways to handle it. We looked at:

- What arithmetic overflow and underflow are
- Integer representations:
 - Unsigned
 - Two's complement
- Ways to deal with overflow/underflow:
 - Exceptions
 - Wrap-around
 - Saturation
- How to detect overflow/underflow
- Implementing a simple Integer wrapper that performs safe arithmetic

[1] An example of how an attacker can exploit integer overflow is the following [SSH1 vulnerability](#).

D2- Module 02 - Hash Tables I



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/18e22e25-8fb5-4763-b92e-ce3ac0d3e4e4/Untitled.png>

Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a hash table

Overview

Hash tables are also called hash maps, maps, unordered maps, or dictionaries. A hash table is a structure that maps keys to values. This makes them extremely efficient for lookups because if you have the key, retrieving the associated value is a constant-time operation.

Follow Along

Time and Space Complexity

Lookup

Hash tables have fast lookups ($O(1)$) on average. However, in the worst case, they have slow ($O(n)$) lookups. The slow lookups happen when there is a hash collision (two different keys hash to the same index).

Insert

Hash tables have fast insertions ($O(1)$) on average. However, in the worst case, they have slow ($O(n)$) insertions. Just like with the lookups, the worst case occurs due to hash collisions.

Delete

Hash tables have fast deletes ($O(1)$) on average. However, in the worst case, they have slow ($O(n)$) deletions. Just like with lookups and insertions, the worst case occurs due to hash collisions.

Space

The space complexity of a hash table is linear ($O(n)$). Each key-value pair in the hash table will take up space in memory.

Strengths

The main reason why hash tables are great is that they have constant-time ($O(1)$) lookup operations in the average case. That makes them great to use in any situation where you will be conducting many lookup operations. The second reason they are great is that they allow you to use any hashable object as a key. This means they can be used in many different scenarios where you want to map one object (the key) to another object (the value).

Weaknesses

One weakness of the hash table is that the mapping goes only one way. So, if you know the key, it's incredibly efficient to retrieve the mapped value to that key. However, if you know the value and want to find the key that is mapped to that value, it is inefficient. Another weakness is that if your hash function produces lots of collisions, the hash table's time complexity gets worse and worse. This is because the underlying linked lists are inefficient for lookups.

What About Hash Collisions?

A hash collision is when our hash function returns the same index given two different keys. Theoretically, there is no perfect hash function, though some are better than others. Thus, any hash table implementation has to have a strategy to deal with the scenario when two different keys hash to the same index. You can't just have the hash table overwrite the already existing value.

The most common strategy for dealing with hash collisions is not storing the values directly at an index of the hash table's array. Instead, the array index stores a pointer to a linked list. Each node in the linked list stores a key, value, and a pointer to the next item in the linked list.

The above is just one of the ways to deal with hash collisions. Hopefully, you can now see why all of our hash table operations become $O(n)$ in the worst case. What is the worst case? The worst case is when all of the keys collide at the same index in our hash table. If we have ten items in our hash table, all ten items are stored in one linked list at the same index of our array. That means that our hash table has the same efficiency as a linked list in the worst case.

Challenge

1. In your own words, explain how and why the time complexity of hash table operations degrades to $O(n)$ in the worst case.

Additional Resources

- <https://www.geeksforgeeks.org/hashing-data-structure/> (Links to an external site.)

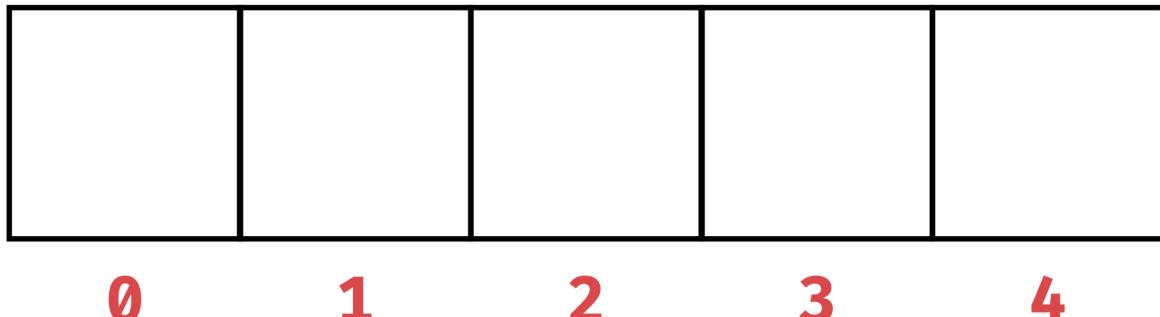


<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/18e22e25-8fb5-4763-b92e-ce3ac0d3e4e4/Untitled.png>

Objective 02 - Describe and implement a hash function

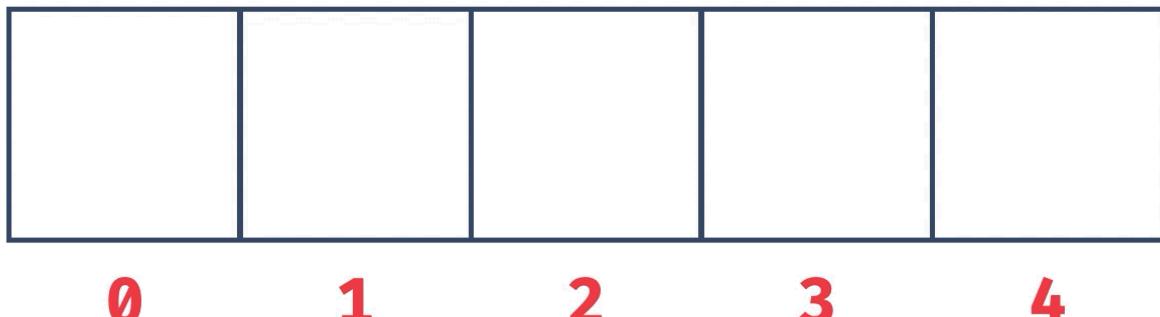
Overview

Hashing functions take an input (usually a string) and return an integer as the output. Let's say we needed to store five colors in our hash table. Currently, we have an empty table that looks like this:



https://tk-assets.lambdaschool.com/add0f486-f742-4b70-9885-88c6938237f8_Untitled.png

Now, I need to assign an index given the name of a color. Our hash function will take the name of a color and convert it into an index.



https://tk-assets.lambdaschool.com/16439e40-5ec9-4242-b5c5-07584bc665ca_S5-M1-01-Hash-Table-Animation.gif

So, hash functions convert the strings into indexes, and then we store the given string into the computed index of an array.

Hash Function + Array = Hash Table

Now that we know what a hash table is let's dive deeper into creating a hash function.

Follow Along

To convert a string into an integer, hashing functions operate on the individual characters that make up the string.

Let's use what we know to create a hashing function in Python.

In Python, we can encode strings into their bytes representation with the `.encode()` method (read more [here \(Links to an external site.\)](#)). Once encoded, an integer represents each character.

Let's do this with the string `hello`

```
1 bytes_representation = "hello".encode()
2
3 for byte in bytes_representation:
4     print(byte)
5
6 ### Print Output
7 ### 104
8 ### 101
9 ### 108
10 ### 108
11 ### 111
```

Now that we've converted our string into a series of integers, we can manipulate those integers somehow. For simplicity's sake, we can use a simple accumulator pattern to get a sum of all the integer values.

```
1 bytes_representation = "hello".encode()
2
3 sum = 0
4 for byte in bytes_representation:
5     sum += byte
6
7 print(sum)
8
9 ### Print Output
10 ### 532
```

Great! We turned a string into a number. Now, let's generalize this into a function.

```
1 def my_hashing_func(str):
2     bytes_representation = str.encode()
3
4     sum = 0
5     for byte in bytes_representation:
6         sum += byte
7
8     return sum
```

We aren't done yet ☹. As shown earlier, `hello` returns `532`. But, what if our hash table only has ten slots? We have to make 532 a number less than 10 ☹.

Remember the modulo operator `%`? We can use that in our hashing function to ensure that the integer the function returns is within a specific range.

```
1 def my_hashing_func(str, table_size):
2     bytes_representation = str.encode()
```

```
3
4     sum = 0
5     for byte in bytes_representation:
6         sum += byte
7
8     return sum % table_size
```

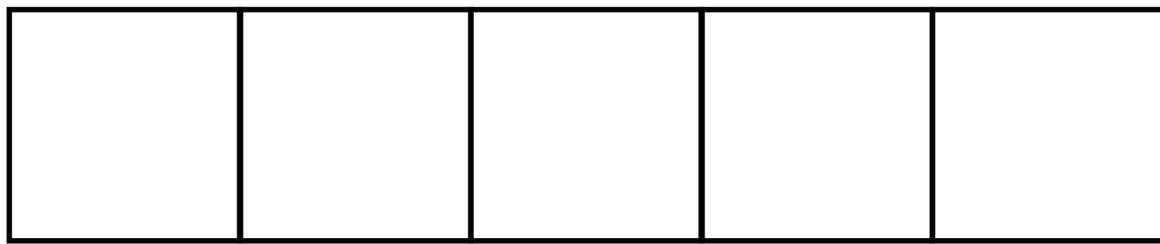


<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/18e22e25-8fb5-4763-b92e-ce3ac0d3e4e4/Untitled.png>

Objective 02 - Describe and implement a hash function

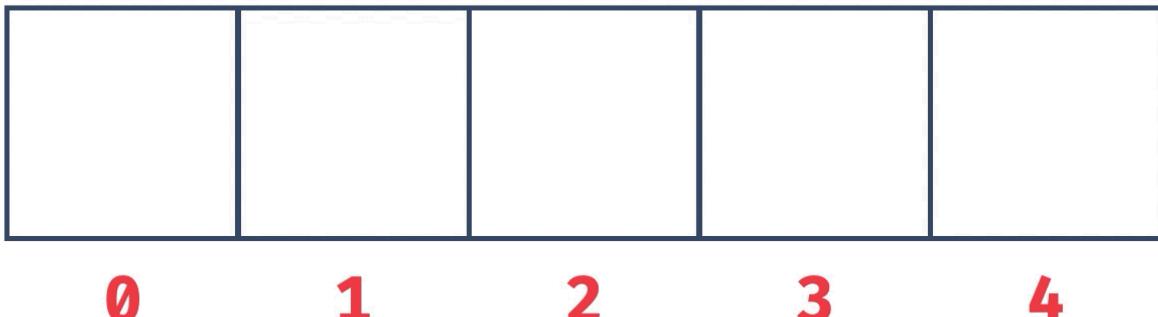
Overview

Hashing functions take an input (usually a string) and return an integer as the output. Let's say we needed to store five colors in our hash table. Currently, we have an empty table that looks like this:



https://tk-assets.lambdaschool.com/add0f486-f742-4b70-9885-88c6938237f8_Untitled.png

Now, I need to assign an index given the name of a color. Our hash function will take the name of a color and convert it into an index.



https://tk-assets.lambdaschool.com/16439e40-5ec9-4242-b5c5-07584bc665ca_S5-M1-01-Hash-Table-Animation.gif

So, hash functions convert the strings into indexes, and then we store the given string into the computed index of an array.

Hash Function + Array = Hash Table

Now that we know what a hash table is let's dive deeper into creating a hash function.

Follow Along

To convert a string into an integer, hashing functions operate on the individual characters that make up the string.

Let's use what we know to create a hashing function in Python.

In Python, we can encode strings into their bytes representation with the `.encode()` method (read more [here \(Links to an external site.\)](#)). Once encoded, an integer represents each character.

Let's do this with the string `hello`

```

1 bytes_representation = "hello".encode()
2
3 for byte in bytes_representation:
4     print(byte)
5
6 ### Print Output
7 ### 104
8 ### 101
9 ### 108
10 ### 108
11 ### 111

```

Now that we've converted our string into a series of integers, we can manipulate those integers somehow. For simplicity's sake, we can use a simple accumulator pattern to get a sum of all the integer values.

```
1 bytes_representation = "hello".encode()
2
3 sum = 0
4 for byte in bytes_representation:
5     sum += byte
6
7 print(sum)
8
9 ### Print Output
10 ### 532
```

Great! We turned a string into a number. Now, let's generalize this into a function.

```
1 def my_hashing_func(str):
2     bytes_representation = str.encode()
3
4     sum = 0
5     for byte in bytes_representation:
6         sum += byte
7
8     return sum
```

We aren't done yet ☹. As shown earlier, `hello` returns `532`. But, what if our hash table only has ten slots? We have to make 532 a number less than 10 ☹.

Remember the modulo operator `%`? We can use that in our hashing function to ensure that the integer the function returns is within a specific range.

```
1 def my_hashing_func(str, table_size):
2     bytes_representation = str.encode()
3
4     sum = 0
5     for byte in bytes_representation:
6         sum += byte
7
8     return sum % table_size
```

<https://replit.com/@bgoonz/cs-unit-1-sprint-4-module-1-hash-function#main.py>



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/18e22e25-8fb5-4763-b92e-ce3ac0d3e4e4/Untitled.png>

Objective 03 - Implement a user-defined HashTable class that allows basic operations

Overview

We define a hash table as an empty array and hash function as a function that takes a value and converts it into an array index where you will store that value. Let's put the two together. Let's implement a `HashTable` class where we can:

- Insert values into a hash table
- Retrieve values from a hash table
- Delete values from a hash table

Let's start with the insert function. For an insert, I need to insert a value with an associated key. Let's store the instructors at Lambda and where they live. We want to store:

- ("Parth", "California")
- ("Beej", "Oregon")
- ("Dustin", "Utah")
- ("Ryan", "Utah")

Here's what our `HashTable` class looks like right now:

```
1  class HashTable:  
2      """  
3          A hash table with `capacity` buckets  
4          that accepts string keys  
5          """  
6  
7      def __init__(self, capacity):  
8          self.capacity = capacity # Number of buckets in the hash table  
9          self.storage = [None] * capacity  
10         self.item_count = 0  
11  
12     def get_num_slots(self):  
13         """  
14             Return the length of the list you're using to hold the hash table data. (Not the number of items stored in  
15             but the number of slots in the main list.)  
16             One of the tests relies on this.  
17             """  
18  
19         return len(self.storage)  
20  
21     def djb2(self, key):  
22         """  
23             DJB2 hash, 32-bit  
24             """  
25  
26         # Cast the key to a string and get bytes  
27         str_key = str(key).encode()  
28  
29         # Start from an arbitrary large prime  
30         hash_value = 5381  
31  
32         # Bit-shift and sum value for each character  
33         for b in str_key:  
34             hash_value = ((hash_value << 5) + hash_value) + b  
35             hash_value &= 0xffffffff # DJB2 is a 32-bit hash, only keep 32 bits  
36  
37     def hash_index(self, key):  
38         """  
39             Take an arbitrary key and return a valid integer index within the hash table's storage capacity.  
40             """  
41  
42         return self.djb2(key) % self.capacity  
43  
44     def put(self, key, value):  
45         """  
46             Store the value with the given key.  
47             """  
48  
49     def delete(self, key):  
50         """  
51             Remove the value stored with the given key.  
52             Print a warning if the key is not found.  
53             """  
54  
55     def get(self, key):  
56         """  
57             Retrieve the value stored with the given key.  
58             Returns None if the key is not found.  
59             """
```

Let's break this down a little bit. Our `init` function takes in the length of a hash table and creates an empty array. Our `hash_index` function takes a key and computes and index using the famous `djb2` hash function. Let's implement the other functions (`put`, `delete`, `get`).

Follow Along

The `put` Method

Let's create our `put` function. Before we code, let's break down what needs to happen:

- Given a `key` and a `value`, insert the respective `value` into a hash table array using the hashed `key` to determine the storage location index.

Let's think about what we need to do:

- Hash the `key` into an index using the hash function
- Put the `value` into that index

You might be thinking, "What if two keys hash to the same index?" That's a great question, and we will worry about that later. It's a nifty solution ☺. But for now, let's worry about hashing a key and storing a value.

First, let's call the hash function and store the return value in `index`:

```
1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     """
5     index = self.hash_index(key)
```

Next, let's insert the value at that index:

```
1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     """
5     index = self.hash_index(key)
6     self.storage[index] = value
7     return
```

There we go! Given a key, we hashed it and inserted a value. Again, we will worry about colliding indices later ☺.

The `delete` Method

Next, let's write our `delete` method. What does this method need to do? We can think about it as the inverse of the `put` method that we just defined. The function will receive a `key` as its input, then pass that `key` through the hash function to get the index where the hash table's value needs to be deleted.

Let's start by getting the index by passing the `key` through the hashing function:

```
1 def delete(self, key):
2     """
3     Remove the value stored with the given key.
4     """
5     index = self.hash_index(key)
```

Next, we need to delete the value from that index in our storage by setting it to `None`. Remember, we aren't dealing with collisions in this example. If we had to deal with collisions, this would be more complex.

```
1 def delete(self, key):
2     """
3     Remove the value stored with the given key.
4     """
5     index = self.hash_index(key)
6     self.storage[index] = None
```

The `get` Method

The last method we need to deal with is our `get` method. `get` is a simple method that retrieves the `value` stored at a specific `key`. The function needs to receive a `key` as an input, pass that `key` through the hashing function to find the index where the value is stored, and then return the `value` at that index.

Let's start by getting the index from the `key`:

```
1 def get(self, key):
2     """
3     Retrieve the value stored with the given key.
4     Returns None if the key is not found.
5     """
6     index = self.hash_index(key)
```

Next, we need to return the value that is stored at the `index`.

```
1 def get(self, key):
2     """
3     Retrieve the value stored with the given key.
4     Returns None if the key is not found.
5     """
6     index = self.hash_index(key)
7     return self.storage[index]
```

D3-Module 03 - Hash Tables II

Original:

 Google Colaboratory

<https://colab.research.google.com/drive/1WXURLnQJopWW5J-OKxOePd4GTeDM542p?usp=sharing#scrollTo=Um92huhOx2BD>

 Number Bases and Chars.ipynb

<https://gist.github.com/bgoonz/85cf385ba5382cea548c2b6083cd1b3f>

 HT2.ipynb

<https://gist.github.com/bgoonz/c10af728179ff056894c6f17dfb819bc#file-ht2-ipynb>

Objective 01 - Understand hash collisions and use a linked list for collision resolution in a user-defined Hashable class

Overview

Remember when we wondered what would happen if multiple keys hashed to the same index, and we said that we would worry about it later? Whelp, it's later ☐.

Let's say we were given the key-value pair `("Ryan", 10)`. Our hash code then maps "Ryan" to index 3. Excellent, that works! Now let's say after we inserted `("Ryan", 10)`, we have to insert `("Parth", 12)`. Our hash code maps "Parth" to index 3. Uh oh! Ryan is already there! What do we do?? ☐

Ok, let's stop freaking out, and let's think about this. If we don't do anything, the value stored at index 3 will just get overwritten. Meaning if we try to retrieve the value associated with `"Ryan"`, 12 will be returned instead of 10. That might not seem like a big deal, but what if we were returning passwords based on a user ID, and we returned someone else's password. That would be horrible.

Let's fix this problem. The most common way to solve this is with **chaining**. If we see multiple values hashed to an index, we will chain them in a some data structure that can hold multiple items. In our case, we'll use Python's `list` type, but a more typical solution would use a linked list. We'll cover linked lists in a future module.

https://tk-assets.lambdaschool.com/f952600c-f3e0-4d96-bb53-def08235c9c0_collision.gif

Ok, sounds ideal? But how does this work in code? Let's write some of it together.

Follow Along

Below is a partially filled out hash table class where we will be using `HashTableEntry` as our chain entries.

Take a look at the code below.

```
1  class HashTableEntry:
2      """
3          Hash table key/value pair to go in our collision chain
4      """
5      def __init__(self, key, value):
6          self.key = key
7          self.value = value
8
9      # Hash table can't have fewer than this many slots
10     MIN_CAPACITY = 8
11
12     class HashTable:
13         """
14             A hash table with `capacity` buckets
15             that accepts string keys
16             Implement this.
17         """
18
19         def __init__(self, capacity):
20             self.capacity = capacity # Number of buckets in the hash table
21
22             self.storage = []
23             for _ in range(capacity): # Initialize with empty lists
24                 self.storage.append([])
25
26             self.item_count = 0
27
28         def get_num_slots(self):
29             """
30                 Return the length of the list you're using to hold the hash table data. (Not the number of items stored in
31                 but the number of slots in the main list.)
```

```

32     One of the tests relies on this.
33     Implement this.
34     """
35     # Your code here
36
37     def get_load_factor(self):
38         """
39             Return the load factor for this hash table.
40             Implement this.
41             """
42             return len(self.storage)
43
44     def djb2(self, key):
45         """
46             DJB2 hash, 32-bit
47             Implement this, and/or FNV-1.
48             """
49             str_key = str(key).encode()
50
51             hash = FNV_offset_basis_64
52
53             for b in str_key:
54                 hash *= FNV_prime_64
55                 hash ^= b
56                 hash &= 0xffffffffffffffff # 64-bit hash
57
58             return hash
59
60     def hash_index(self, key):
61         """
62             Take an arbitrary key and return a valid integer index between within the hash table's storage capacity.
63             """
64             return self.djb2(key) % self.capacity
65
66     def put(self, key, value):
67         """
68             Store the value with the given key.
69             Hash collisions should be handled with Linked List Chaining.
70             Implement this.
71             """
72             # Your code here
73
74     def delete(self, key):
75         """
76             Remove the value stored with the given key.
77             Print a warning if the key is not found.
78             Implement this.
79             """
80             # Your code here
81
82     def get(self, key):
83         """
84             Retrieve the value stored with the given key.
85             Returns None if the key is not found.
86             Implement this.
87             """
88             # Your code here

```

Let's implement the `put` method with collision resolution by chaining. What are the two cases we need to handle?

1. **There are no entries at the index.** Great! We can initialize the entry to a list with the new `HashTableEntry` in it.
2. **There are multiple entries at the index.** We need to check every entry in the chain. If the key in one of the entries is equal to the key we are passing in, we need to replace it. For instance, let's say we pass in `("Ryan", 12)`, and then we later pass in `("Ryan", 15)`. We would need to replace "Ryan"s old value with 15. If there are no entries that match, we create a new entry at the end of the chain.

Ok, that might sound confusing. Let's start breaking it down into code.

```

1     def put(self, key, value):
2         """

```

```
3     Store the value with the given key.  
4     Hash collisions should be handled with Linked List Chaining.  
5     Implement this.  
6     """  
7     # Your code here
```

First, we need to hash the key and start with the first entry at that index.

```
1  def put(self, key, value):  
2      """  
3          Store the value with the given key.  
4          Hash collisions should be handled with Linked List Chaining.  
5          Implement this.  
6          """  
7          index = self.hash_index(key)  
8  
9          chain = self.storage[index]
```

Next, we need to go through the chain. We need to check two conditions:

1. The current entry is not empty.
2. The key or the current entry is not equal to the key we are passing in.

```
1  def put(self, key, value):  
2      """  
3          Store the value with the given key.  
4          Hash collisions should be handled with Linked List Chaining.  
5          Implement this.  
6          """  
7          index = self.hash_index(key)  
8  
9          chain = self.storage[index]  
10  
11         existing_entry = None  
12  
13         for current_entry in chain:  
14             if current_entry.key == key:  
15                 existing_entry = current_entry  
16                 break
```

Sweet! Now we need to check what happens when the loop breaks. It would only break for two reasons:

1. We reached an entry with the same key and need to replace the value.
2. We reached the end of the chain and need to create a new entry.

Let's write that in code!

```
1  def put(self, key, value):  
2      """  
3          Store the value with the given key.  
4          Hash collisions should be handled with Linked List Chaining.  
5          Implement this.  
6          """  
7          index = self.hash_index(key)  
8  
9          chain = self.storage[index]  
10  
11         existing_entry = None  
12  
13         for current_entry in chain:  
14             if current_entry.key == key:
```

```
15         existing_entry = current_entry
16         break
17
18     if existing_entry is not None:
19         existing_entry.value = value
20     else:
21         new_entry = HashTableEntry(key, value)
22         chain.append(new_entry)
```

Great! We created the `put` method.

Challenge

<https://replit.com/@bgoonz/cs-unit-1-sprint-4-module-2-hash-table-collision-resolution#main.py>



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/155e4481-6522-4f77-8cc1-72004e760287/Untitled.png>

Objective 02 - Define and compute the load factor of a hash table and implement a hash table that automatically resizes based on load factor

Overview

What does runtime look like with linked list chaining?

The performance of hash tables for search, insertion, and deletion is constant time ($O(1)$) in the average case. However, as the chains get longer and longer, in the worst case, those same operations are done in linear time ($O(n)$). The more collisions that your hash table has, the less performant the hash table is. To avoid collisions, a proper hash function and maintaining a low load factor is crucial. What is a load factor?

Load Factor

The load factor of a hash table is trivial to calculate. You take the number of items stored in the hash table divided by the number of slots.

$$\text{Load Factor} = \frac{\text{Number of Items in Hash Table}}{\text{Total Number of Slots}}$$

https://tk-assets.lambdaschool.com/59d00218-52e2-4f3d-9680-2b2d8baad3ae_S5-M3-O1LoadFactor.001.jpeg

Hash tables use an array for storage. So, the load factor is the number of occupied slots divided by the length of the array. So, an array of length 10 with three items in it has a load factor of 0.3, and an array of length 20 with twenty items has a load factor of 1. If you use linear probing for collision resolution, then the maximum load factor is 1. If you use chaining for collision resolution, then the load factor can be greater than 1.

As the load factor of your hash table increases, so does the likelihood of a collision, which reduces your hash table's performance. Therefore, you need to monitor the load factor and resize your hash table when the load factor gets too large. The general rule of thumb is to resize your hash table when your load factor is greater than 0.7. Also, when you resize, it is common to double the size of the hash table. When you resize the array, you need to re-insert all of the items into this new hash table. You cannot simply copy the old items into the new hash table. Each item has to be rerun through the hashing function because the hashing function considers the size of the hash table when determining the index that it returns.

You can see that resizing is an expensive operation, so you don't want to resize too often. However, when we average it out, hash tables are constant time ($O(1)$) even with resizing.

The load factor can also be too small. If the hash table is too large for the data that it is storing, then memory is being wasted. So, in addition to resizing, when the load factor gets too high, you should also resize when the load factor gets too low.

One way to know when to resize your hash table is to compute the load factor whenever an item is inserted or deleted into the hash table. If the load factor is too high or too low, then you need to resize.

We added a `get_load_factor` and `resize` method to calculate the load factor and resize the hash table with a new capacity when necessary.

```

1 class HashTableEntry:
2     """
3         Linked List hash table key/value pair
4     """
5     def __init__(self, key, value):
6         self.key = key
7         self.value = value
8         self.next = None
9
10 # Hash table can't have fewer than this many slots

```

```

11 MIN_CAPACITY = 8
12
13 class HashTable:
14     """
15     A hash table with `capacity` buckets
16     that accepts string keys
17     Implement this.
18     """
19
20     def __init__(self, capacity):
21         self.capacity = capacity # Number of buckets in the hash table
22         self.storage = [None] * capacity
23         self.item_count = 0
24
25     def get_num_slots(self):
26         """
27             Return the length of the list you're using to hold the hash
28             table data. (Not the number of items stored in the hash table,
29             but the number of slots in the main list.)
30             One of the tests relies on this.
31             Implement this.
32             """
33         # Your code here
34
35     def get_load_factor(self):
36         """
37             Return the load factor for this hash table.
38             Implement this.
39             """
40         return self.item_count / self.capacity
41
42     def resize(self, new_capacity):
43         """
44             Changes the capacity of the hash table and
45             rehashes all key/value pairs.
46             Implement this.
47             """
48         old_storage = self.storage
49         self.capacity = new_capacity
50         self.storage = [None] * self.capacity
51
52         current_entry = None
53
54         # Save this because put adds to it, and we don't want that.
55         # It might be less hackish to pass a flag to put indicating that
56         # we're in a resize and don't want to modify item count.
57         old_item_count = self.item_count
58
59         for bucket_item in old_storage:
60             current_entry = bucket_item
61             while current_entry is not None:
62                 self.put(current_entry.key, current_entry.value)
63                 current_entry = current_entry.next
64
65         # Restore this to the correct number
66         self.item_count = old_item_count
67
68     def djb2(self, key):
69         """
70             DJB2 hash, 32-bit
71             Implement this, and/or FNV-1.
72             """
73         str_key = str(key).encode()
74
75         hash = FNV_offset_basis_64
76
77         for b in str_key:
78             hash *= FNV_prime_64
79             hash ^= b
80             hash &= 0xffffffffffffffff # 64-bit hash
81
82         return hash
83
84     def hash_index(self, key):
85         """
86             Take an arbitrary key and return a valid integer index

```

```

87     within the hash table's storage capacity.
88     """
89     return self.djb2(key) % self.capacity
90
91     def put(self, key, value):
92         """
93             Store the value with the given key.
94             Hash collisions should be handled with Linked List Chaining.
95             Implement this.
96             """
97             index = self.hash_index(key)
98
99             current_entry = self.storage[index]
100
101            while current_entry is not None and current_entry.key != key:
102                current_entry = current_entry.next
103
104            if current_entry is not None:
105                current_entry.value = value
106            else:
107                new_entry = HashTableEntry(key, value)
108                new_entry.next = self.storage[index]
109                self.storage[index] = new_entry
110
111        def delete(self, key):
112            """
113                Remove the value stored with the given key.
114                Print a warning if the key is not found.
115                Implement this.
116                """
117                # Your code here
118
119        def get(self, key):
120            """
121                Retrieve the value stored with the given key.
122                Returns None if the key is not found.
123                Implement this.
124                """
125                # Your code here

```

Follow Along

Let's change our `put` method to resize when the load factor gets too high. Here's how our current `put` method looks:

```

1  def put(self, key, value):
2      """
3          Store the value with the given key.
4          Hash collisions should be handled with Linked List Chaining.
5          Implement this.
6          """
7          index = self.hash_index(key)
8
9          current_entry = self.storage[index]
10
11         while current_entry is not None and current_entry.key != key:
12             current_entry = current_entry.next
13
14         if current_entry is not None:
15             current_entry.value = value
16         else:
17             new_entry = HashTableEntry(key, value)
18             new_entry.next = self.storage[index]
19             self.storage[index] = new_entry

```

To know when to resize, we need to correctly increment the count whenever we insert something new into the hash table. Let's go ahead and add that.

```

1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     Hash collisions should be handled with Linked List Chaining.
5     Implement this.
6     """
7     index = self.hash_index(key)
8
9     current_entry = self.storage[index]
10
11    while current_entry is not None and current_entry.key != key:
12        current_entry = current_entry.next
13
14    if current_entry is not None:
15        current_entry.value = value
16    else:
17        new_entry = HashTableEntry(key, value)
18        new_entry.next = self.storage[index]
19        self.storage[index] = new_entry
20
21    self.item_count += 1

```

Next, we need to check if the load factor is greater than or equal to 0.7. If it is, we need to double our capacity and resize.

```

1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     Hash collisions should be handled with Linked List Chaining.
5     Implement this.
6     """
7     index = self.hash_index(key)
8
9     current_entry = self.storage[index]
10
11    while current_entry is not None and current_entry.key != key:
12        current_entry = current_entry.next
13
14    if current_entry is not None:
15        current_entry.value = value
16    else:
17        new_entry = HashTableEntry(key, value)
18        new_entry.next = self.storage[index]
19        self.storage[index] = new_entry
20
21    self.item_count += 1
22
23    if self.get_load_factor() > 0.7:
24        self.resize(self.capacity * 2)

```

Fantastic, we did it!



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/18e22e25-8fb5-4763-b92e-ce3ac0d3e4e4/Untitled.png>

Objective 02 - Define and compute the load factor of a hash table and implement a hash table that automatically resizes based on load factor

Overview

What does runtime look like with linked list chaining?

The performance of hash tables for search, insertion, and deletion is constant time ($O(1)$) in the average case. However, as the chains get longer and longer, in the worst case, those same operations are done in linear time ($O(n)$). The more collisions that your hash table has, the less performant the hash table is. To avoid collisions, a proper hash function and maintaining a low load factor is crucial. What is a load factor?

Load Factor

The load factor of a hash table is trivial to calculate. You take the number of items stored in the hash table divided by the number of slots.

$$\text{Load Factor} = \frac{\text{Number of Items in Hash Table}}{\text{Total Number of Slots}}$$

https://tk-assets.lambdaschool.com/59d00218-52e2-4f3d-9680-2b2d8baad3ae_S5-M3-O1LoadFactor.001.jpeg

Hash tables use an array for storage. So, the load factor is the number of occupied slots divided by the length of the array. So, an array of length 10 with three items in it has a load factor of 0.3, and an array of length 20 with twenty items has a load factor of 1. If you use linear probing for collision resolution, then the maximum load factor is 1. If you use chaining for collision resolution, then the load factor can be greater than 1.

As the load factor of your hash table increases, so does the likelihood of a collision, which reduces your hash table's performance. Therefore, you need to monitor the load factor and resize your hash table when the load factor gets too large. The general rule of thumb is to resize your hash table when your load factor is greater than 0.7. Also, when you resize, it is common to double the size of the hash table. When you resize the array, you need to re-insert all of the items into this new hash table. You cannot simply copy the old items into the new hash table. Each item has to be run through the hashing function because the hashing function considers the size of the hash table when determining the index that it returns.

You can see that resizing is an expensive operation, so you don't want to resize too often. However, when we average it out, hash tables are constant time ($O(1)$) even with resizing.

The load factor can also be too small. If the hash table is too large for the data that it is storing, then memory is being wasted. So, in addition to resizing, when the load factor gets too high, you should also resize when the load factor gets too low.

One way to know when to resize your hash table is to compute the load factor whenever an item is inserted or deleted into the hash table. If the load factor is too high or too low, then you need to resize.

We added a `get_load_factor` and `resize` method to calculate the load factor and resize the hash table with a new capacity when necessary.

```
1 class HashTableEntry:
2     """
3     Linked List hash table key/value pair
4     """
5     def __init__(self, key, value):
6         self.key = key
7         self.value = value
8         self.next = None
9
10    # Hash table can't have fewer than this many slots
11    MIN_CAPACITY = 8
12
13    class HashTable:
14        """
15        A hash table with `capacity` buckets
16        that accepts string keys
17        Implement this.
18        """
19
20        def __init__(self, capacity):
21            self.capacity = capacity # Number of buckets in the hash table
22            self.storage = [None] * capacity
23            self.item_count = 0
24
25        def get_num_slots(self):
26            """
27            Return the length of the list you're using to hold the hash
28            table data. (Not the number of items stored in the hash table,
29            but the number of slots in the main list.)
30            One of the tests relies on this.
31            Implement this.
32            """
33            # Your code here
34
35        def get_load_factor(self):
36            """
37            Return the load factor for this hash table.
38            Implement this.
39            """
40            return self.item_count / self.capacity
41
42        def resize(self, new_capacity):
43            """
44            Changes the capacity of the hash table and
45            rehashes all key/value pairs.
46            Implement this.
47            """
48            old_storage = self.storage
49            self.capacity = new_capacity
50            self.storage = [None] * self.capacity
51
52            current_entry = None
53
54            # Save this because put adds to it, and we don't want that.
55            # It might be less hackish to pass a flag to put indicating that
56            # we're in a resize and don't want to modify item count.
57            old_item_count = self.item_count
58
59            for bucket_item in old_storage:
60                current_entry = bucket_item
61                while current_entry is not None:
62                    self.put(current_entry.key, current_entry.value)
63                    current_entry = current_entry.next
64
65            # Restore this to the correct number
66            self.item_count = old_item_count
67
68        def djb2(self, key):
69            """
70            DJB2 hash, 32-bit
71            Implement this, and/or FNV-1.
```

```

72     """
73     str_key = str(key).encode()
74
75     hash = FNV_offset_basis_64
76
77     for b in str_key:
78         hash *= FNV_prime_64
79         hash ^= b
80         hash &= 0xffffffffffffffff # 64-bit hash
81
82     return hash
83
84 def hash_index(self, key):
85     """
86     Take an arbitrary key and return a valid integer index
87     within the hash table's storage capacity.
88     """
89     return self.djb2(key) % self.capacity
90
91 def put(self, key, value):
92     """
93     Store the value with the given key.
94     Hash collisions should be handled with Linked List Chaining.
95     Implement this.
96     """
97     index = self.hash_index(key)
98
99     current_entry = self.storage[index]
100
101    while current_entry is not None and current_entry.key != key:
102        current_entry = current_entry.next
103
104    if current_entry is not None:
105        current_entry.value = value
106    else:
107        new_entry = HashTableEntry(key, value)
108        new_entry.next = self.storage[index]
109        self.storage[index] = new_entry
110
111 def delete(self, key):
112     """
113     Remove the value stored with the given key.
114     Print a warning if the key is not found.
115     Implement this.
116     """
117     # Your code here
118
119 def get(self, key):
120     """
121     Retrieve the value stored with the given key.
122     Returns None if the key is not found.
123     Implement this.
124     """
125     # Your code here

```

Follow Along

Let's change our `put` method to resize when the load factor gets too high. Here's how our current `put` method looks:

```

1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     Hash collisions should be handled with Linked List Chaining.
5     Implement this.
6     """
7     index = self.hash_index(key)
8
9     current_entry = self.storage[index]
10
11    while current_entry is not None and current_entry.key != key:
12        current_entry = current_entry.next

```

```

13     if current_entry is not None:
14         current_entry.value = value
15     else:
16         new_entry = HashTableEntry(key, value)
17         new_entry.next = self.storage[index]
18         self.storage[index] = new_entry
19

```

To know when to resize, we need to correctly increment the count whenever we insert something new into the hash table. Let's go ahead and add that.

```

1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     Hash collisions should be handled with Linked List Chaining.
5     Implement this.
6     """
7     index = self.hash_index(key)
8
9     current_entry = self.storage[index]
10
11    while current_entry is not None and current_entry.key != key:
12        current_entry = current_entry.next
13
14    if current_entry is not None:
15        current_entry.value = value
16    else:
17        new_entry = HashTableEntry(key, value)
18        new_entry.next = self.storage[index]
19        self.storage[index] = new_entry
20
21    self.item_count += 1

```

Next, we need to check if the load factor is greater than or equal to 0.7. If it is, we need to double our capacity and resize.

```

1 def put(self, key, value):
2     """
3     Store the value with the given key.
4     Hash collisions should be handled with Linked List Chaining.
5     Implement this.
6     """
7     index = self.hash_index(key)
8
9     current_entry = self.storage[index]
10
11    while current_entry is not None and current_entry.key != key:
12        current_entry = current_entry.next
13
14    if current_entry is not None:
15        current_entry.value = value
16    else:
17        new_entry = HashTableEntry(key, value)
18        new_entry.next = self.storage[index]
19        self.storage[index] = new_entry
20
21    self.item_count += 1
22
23    if self.get_load_factor() > 0.7:
24        self.resize(self.capacity * 2)

```

Fantastic, we did it!

Challenge

1. Do we need to modify our `delete` and `get` methods to account for the new `get_load_factor` and `resize` methods? Why or why not?

Additional Resources

- <https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf> (Links to an external site.)

D4- Module 04 - Searching and Recursion

Objective 01 - Understand logarithms and recall the common cases where they come up in technical interviews

Overview

What is a logarithm?

Logarithms are a way of looking differently at exponentials. I know that this is a bit of a vague definition, so let's look at an example.

$$\begin{array}{c} 2^5 \\ 2 \times 2 \times 2 \times 2 \times 2 \end{array}$$

What does the mathematical expression above *mean*? It's an abbreviation for the following expression:

$$\begin{array}{c} 2 * 2 * 2 * 2 * 2 \\ 2 * 2 * 2 * 2 * 2 \end{array}$$

What we are looking at above is two different ways to express an object that doubles in size with each iteration.

Another way to think about $2^5 = 32$ is that 2 is the "growth rate" and 5 is the number of times you went through the growth (doubling). 32 is the final result.

Let's look at a few more expressions:

$$\begin{array}{c} 2^5 = 32 \\ 2^5 = 32 \end{array}$$

$$\begin{array}{c} 2^0 = 1 \\ 2^0 = 1 \end{array}$$

$$\begin{array}{c} 2^{-1} = \frac{1}{2} \\ 2^{-1} = \frac{1}{2} \end{array}$$

Now, to begin looking at logarithms, let's rewrite the exponential expressions above in logarithmic form.

$$\begin{array}{c} \log_2 32 = 5 \\ \log_2 32 = 5 \end{array}$$

$$\begin{array}{c} \log_2 1 = 0 \\ \log_2 1 = 0 \end{array}$$

$$\begin{array}{c} \log_2 \frac{1}{2} = 1 \\ \log_2 \frac{1}{2} = 1 \end{array}$$

Notice how we have essentially just moved around different pieces of the expression.

For our first expression,

$$2^5 = 32$$

$$2^{\textcolor{teal}{5}} = 32$$

`2` was the "growth rate", `5` was the "time" spent growing, and `32` was where we ended up. When we rewrite this logarithmically, we have

$$\log_2 32 = 5$$

$$\log_2 \textcolor{teal}{32} = 5$$

In this case, `2` still represents the "growth rate" and `32` still represents where we end up. The `5` also still represents the "time" spent growing.

So, the difference between when we would use a logarithm and when we use exponentiation depends on what factors we know ahead of time. If you know the growth rate and you know how long you are growing, you can use exponentiation (`2^5`) to figure out where you end up (`32`). However, if you know the growth rate and where you end up but do not know the time spent growing, you can use a logarithm (`\log_2 32`) to figure that out.

Logarithms have an *inverse* relationship with exponents, just like division and multiplication have an inverse relationship.

For example, if you know that you have one group of `5` items and you want to identify the total you would have if you had `4` of those groups instead of just one, you could express that with `5 * 4 = 20`. However, if you knew that you had a total of `20` items and you wanted to know how many groups of `5` you could make out your total, you could express that with `20 \ 5 = 4`.

Follow Along

Why should I care? What does this have to do with programming and interview preparation?

In computer science, you often ask questions like "*How many times must `n` be divided in half before we get to one?*" or "*How many times will we halve this collection before the collection has only one item?*" To answer these questions, **you can use logarithms!** Halving is like doubling, so we can say that `\log_2 n` would give us the answer we're seeking.

You will see this come up when analyzing the time complexity of specific algorithms. Any algorithm that doubles or halves a number or collection on each iteration of a loop is likely to have `O(log n)` time complexity. You will see this come up specifically when we talk about binary search and its time complexity. You will also see this come up in specific sorting algorithms (like merge sort). In simple terms, merge sort divides a collection in half and then merges the sorted halves. The fact that the algorithm repeatedly *halves* something is your clue that it includes a logarithm in its time complexity. One last place you're likely to see logarithms come up is with a perfect binary tree. One property of these binary trees is that the number of nodes *doubles* at each level.

Challenge

1. Write a logarithmic expression that is identical to this exponential expression:

$$2^n = 64$$

2. Write an exponential expression that is identical to this logarithmic expression:

$$\log_2 128 = n$$

3. What keywords should you look out for that might alert you that logarithms are involved?

Additional Resources

- <https://www.mathsisfun.com/algebra/logarithms.html> (Links to an external site.)

- <https://www.interviewcake.com/article/python3/logarithms>



Objective 02 - Write a linear search algorithm

Overview

Imagine that I've chosen a random number from 1 to 20. Then, you must guess the number. One approach would be to start picking at 1 and increment your guess by 1 with each guess. If the computer randomly selected 20, then it would take you 20 guesses to get the correct answer. If the computer guessed 1, then you would have the right answer on your very first guess. On average, you will get the correct answer on the 10th or 11th guess.

If the collection we are searching through is random and unsorted, linear search is the most efficient way to search through it. Once we have a sorted list, then there are more efficient approaches to use.

Follow Along

We want to write a simple program to conduct a linear search on a collection of data. Let's write a function that takes a list (`arr`) and an integer (`target`) as its input and returns the integer `idx` where the target is found. If the `target` does not exist in the `arr`, then the function should return `-1`.

```
1 def linear_search(arr, target):
2     # loop through each item in the input array
3     for idx in range(len(arr)):
4         # check if the item at the current index is equal to the target
5         if arr[idx] == target:
6             # return the current index as the match
7             return idx
8     # if we were able to loop through the entire array, the target is not present
9     return -1
```

Challenge



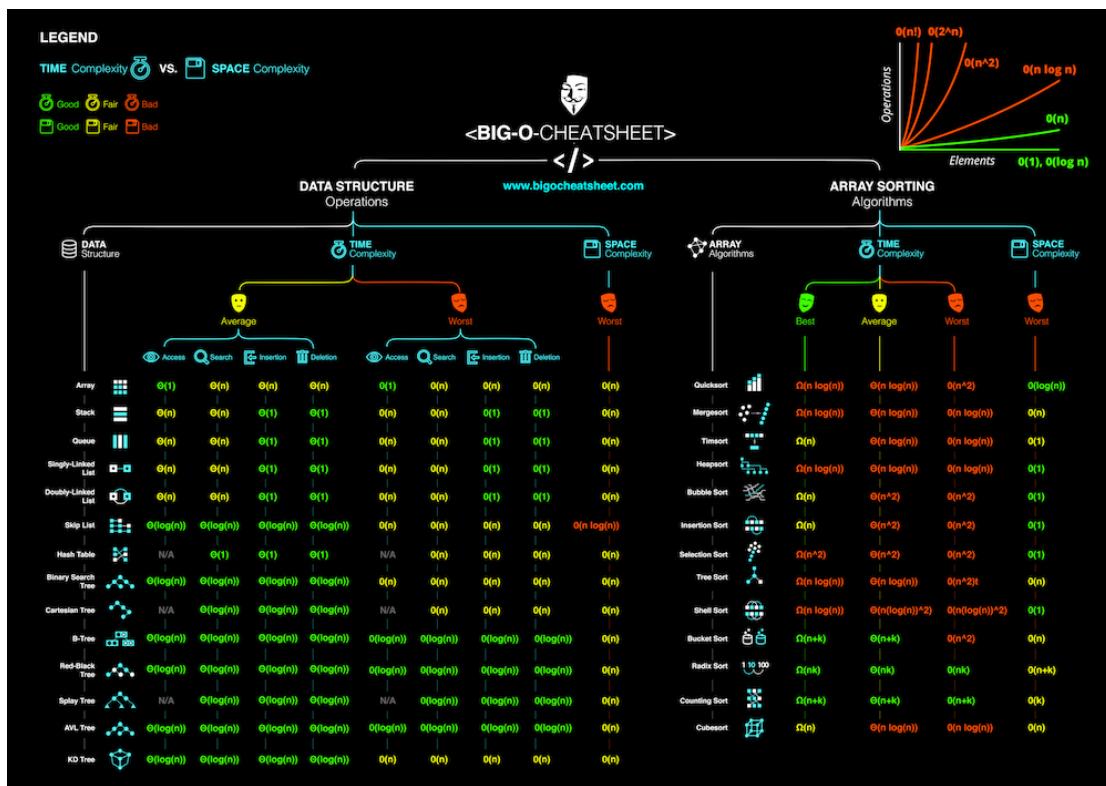
cs-unit-1-sprint-2-module-4-linear-search-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-4-linear-search-2>





wk19



→ D1-Module 01 - Python I

/cirriculum/untitled-3/untitled-2

→ D2- Module 02 - Python II

/cirriculum/untitled-3/untitled-1-1

→ D3- Module 03 - Python III

/cirriculum/untitled-3/untitled-1

→ D4-Module 04 - Python IV

/cirriculum/untitled-3/untitled

→ Overview

/cirriculum/untitled-2/untitled-4

Overview

Overview

During this sprint, we will introduce you to some very common data structures: linked lists, queues, stacks, and binary trees. Additionally, we will teach you about searching through these data structures.

A basic understanding of and the ability to work with these data structures is crucial. These are probably the most common data structures you work with, and an excellent working understanding of them is essential for you to pass a technical interview.

Linked Lists

In this module, you will learn all about linked lists. This a crucial data structure because they form the basis for many other data structures.

Queues and Stacks

This module will teach about queues, stacks, and different implementation options for both. The queue and stack data structures come up frequently during technical interviews and form the basis for necessary traversal techniques that we will look at later.

Binary Search Trees

In this module, you will learn about binary tree properties and binary search trees. These data structures commonly come up during technical interviews, so you need to be comfortable working with them.

Tree Traversal

In this module, you will learn about the different tree traversal methods and practice using them in algorithmic code challenges.

D1- Module 01 - Linked Lists

Objective 01 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a linked list

Overview

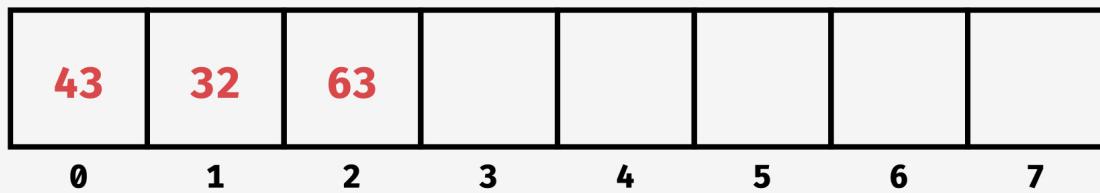
What is a linked list, and how is it different from an array? How efficient or inefficient are its operations? What are its strengths and weaknesses? How can I construct and interact with a linked list? By the end of this objective, you will be able to answer all of these questions confidently.

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.

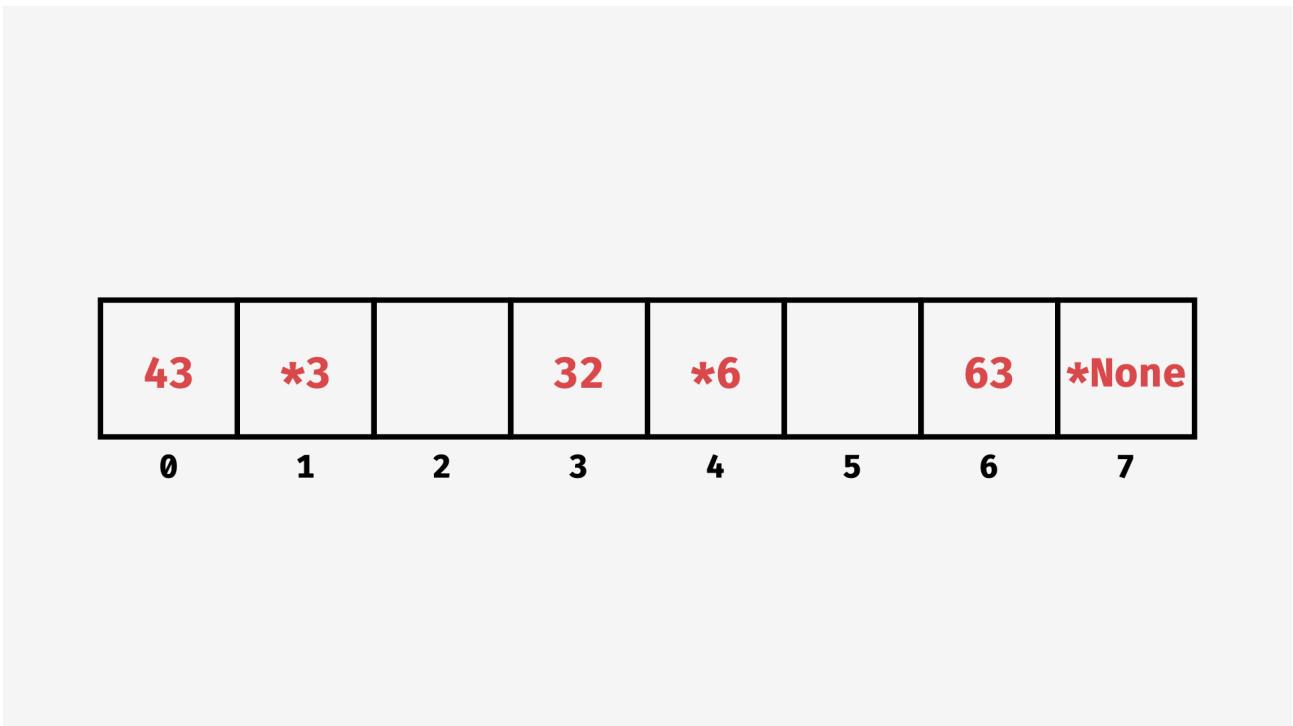
For example, in an array, each element of the list `[43, 32, 63]` is stored in memory like so:



https://tk-assets.lambdaschool.com/61d549f9-9f66-4d1f-9572-2d43098c2767_arrays-stored-in-memory.001.jpeg

`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

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

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.)

 [Linked List Pdf](#)

10-linkedlist.pdf - 960KB

 [Data Structures: Linked Lists](#)

https://www.youtube.com/watch?v=njTh_OwMljA



D2- Module 02 - Queues and Stacks

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 or processes 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.

What are the main findings?

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-3

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-queue-with-linked-list-3>



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"
```



cs-unit-1-sprint-2-module-3-stack-implementation-array-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-stack-implementation-array-2>



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



cs-unit-1-sprint-2-module-3-stack-implementation-linked-li-2

<https://replit.com/@bgoonz/cs-unit-1-sprint-2-module-3-stack-implementation-linked-li-2>



D3- Module 03 - Binary Search Trees

Objective 01 - Describe the properties of a binary tree and the properties of a "perfect" tree

Overview

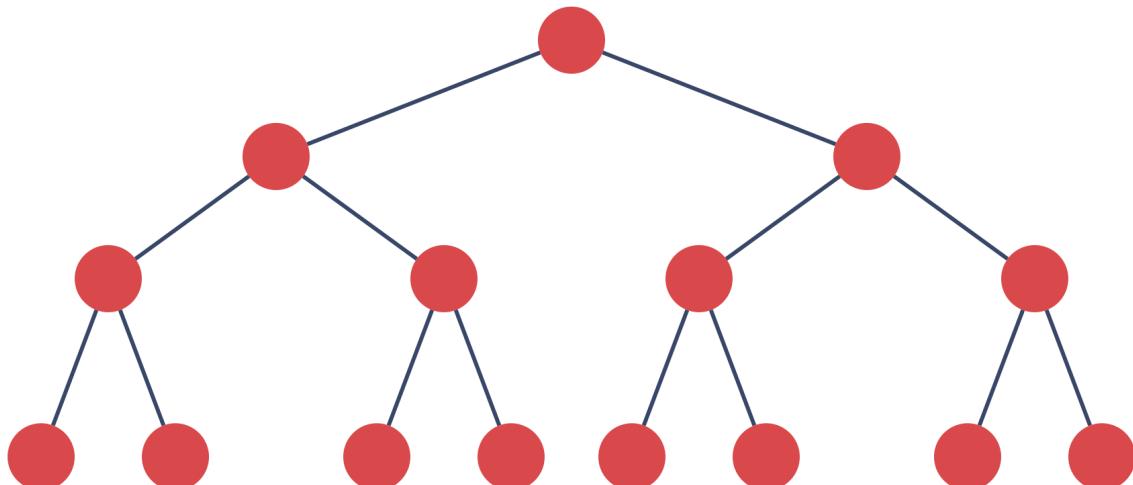
There are lots of different types of tree data structures. A binary tree is a specific type of tree. It is called a binary tree because each node in the tree can only have a maximum of two child nodes. It is common for a node's children to be called either `left` or `right`.

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:

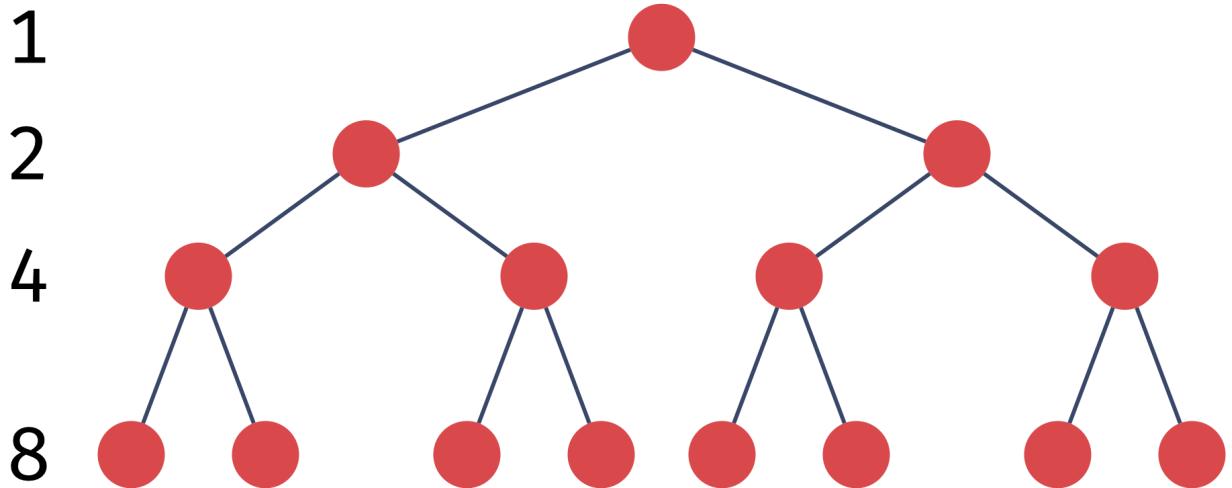


https://tk-assets.lambdaschool.com/c00c8f45-abff-4c3a-b29b-92631b5ac88e_binary-tree-example.001.png

"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 go down.



https://tk-assets.lambdaschool.com/36747e43-d96d-40c9-b8ab-d318f6da8aed_binary-tree-example-levels.001.png

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:

$$\log_2(n + 1) = h$$

[log₂\(n+1\) = 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:

$$n = 2^h - 1$$

[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 (Links to an external site.)
- <https://www.geeksforgeeks.org/binary-tree-data-structure/> (Links to an external site.)

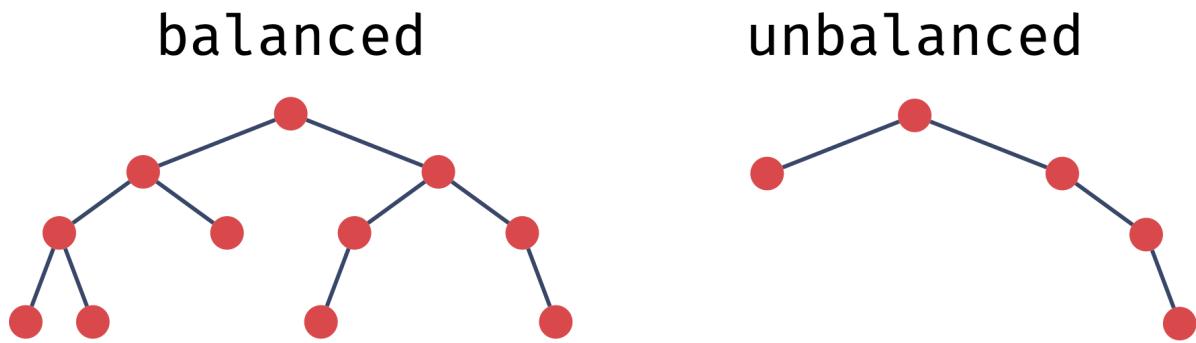


Objective 02 - Recall the time and space complexity, the strengths and weaknesses, and the common uses of a binary search tree

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

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 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 (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
5     def insert(self, value):
6         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
7     def insert(self, value):
8         if value < self.value:
9             if self.left is None:
10                 self.left = BSTNode(value)
11             else:
12                 self.left.insert(value)
13         else:
14             if self.right is None:
15                 self.right = BSTNode(value)
16             else:
17                 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
5     def insert(self, value):
6         self.root.insert(value)
7
8     def search(self, value):
9         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
7     def insert(self, value):
8         if value < self.value:
9             if self.left is None:
10                 self.left = BSTNode(value)
11             else:
12                 self.left.insert(value)
13         else:
14             if self.right is None:
15                 self.right = BSTNode(value)
16             else:
17                 self.right.insert(value)
18
19     def search(self, target):
20         if self.value == target:
21             return self
22         elif target < self.value:
23             if self.left is None:
24                 return False
25             else:
26                 return self.left.search(target)
27         else:
28             if self.right is None:
29                 return False
30             else:
31                 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.)

D4- Module 04 - Tree Traversal

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

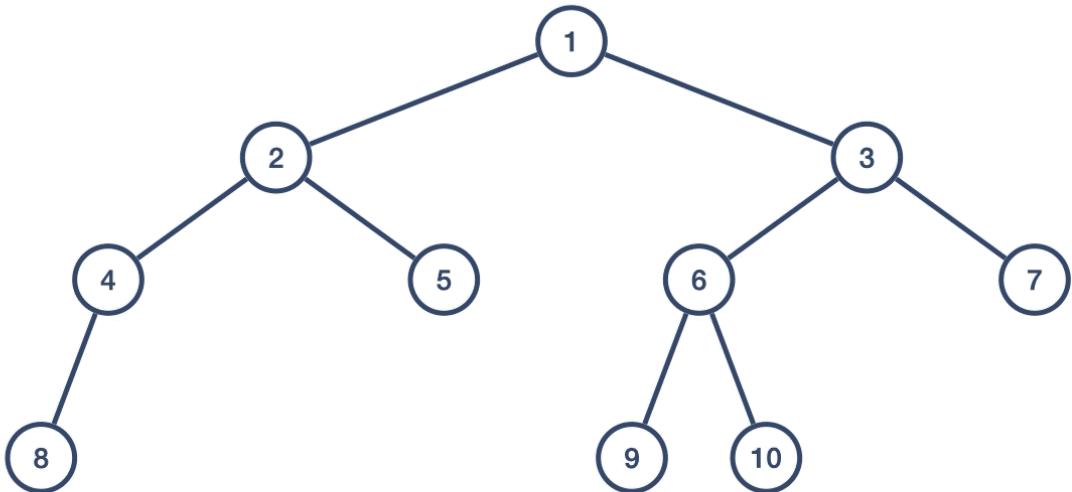
Depth-First Inorder Traversal

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

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
```

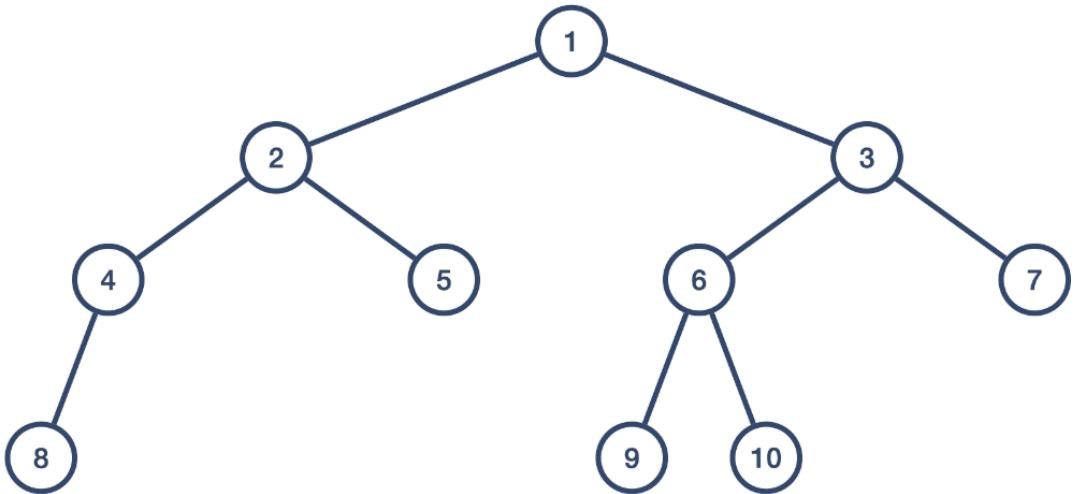
Depth-First Preorder Traversal

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

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(self):
15        result = []
16        self.helper(self, 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.

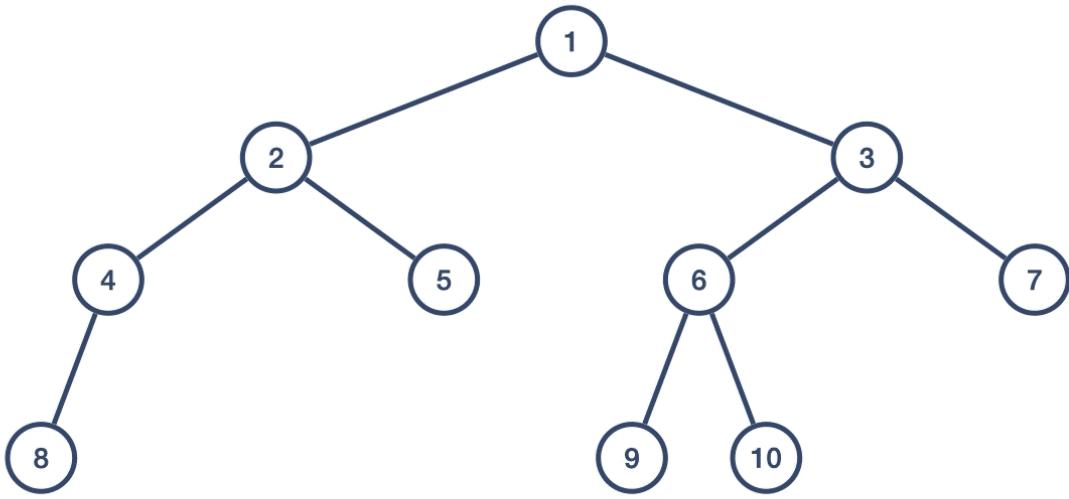
Depth-First Postorder Traversal

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 type of traversal would look.

depth-first postorder traversal



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.

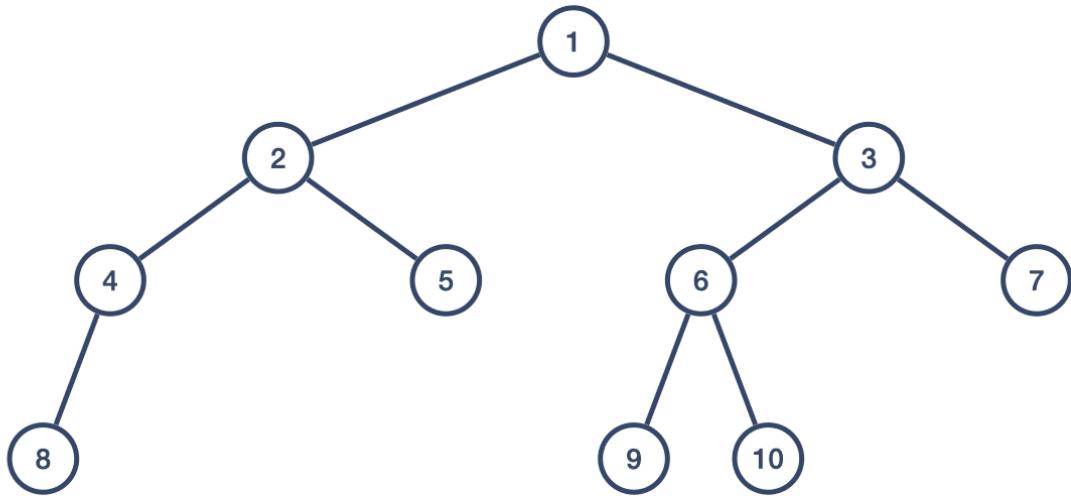
A breadth-first traversal and a level order traversal are the same things. However, a breadth-first traversal can be done on any hierarchical data structure like trees and graphs. But, a level order traversal refers only to the traversal of a tree. Graphs do not have levels like trees do, so that term 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.

breadth-first traversal

queue = []



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(self):
8         if self is None:
9             return []
10
11        result = []
12        queue = []
13        queue.append(self)
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/> (Links to an external site.)



wk20

→ Overview

/cirriculum/untitled-2/untitled-4

→ D1-Module 01 - Python I

/cirriculum/untitled-3/untitled-2

→ D2- Module 02 - Python II

/cirriculum/untitled-3/untitled-1

→ D3- Module 03 - Python III

/cirriculum/untitled-3/untitled-1

→ D4-Module 04 - Python IV

/cirriculum/untitled-3/untitled

Overview

Overview

In this sprint, you will learn about graphs. These data structures are part of the critical computer science fundamentals that interviewers expect you to be comfortable with during the technical interview process.

Additionally, we will teach you about the technical interview process, best practices for interviews, getting unstuck, and overcoming imposter syndrome during an interview. As you complete this sprint and the computer science curriculum, you should feel prepared to tackle the technical interview process successfully.

Graphs I

In this module, you will learn about graph properties, graph representations, and how to build Graph and Vertex classes to construct your graphs in Python.

Graphs II

In this module, you will learn about breadth-first search, depth-first search, and implementing both of these techniques in code.

D1-Graphs I

 CS46 Graphs II.ipynb

<https://gist.github.com/bgoonz/4dc35438f8c293cf68e81c0d73ddfe1a>



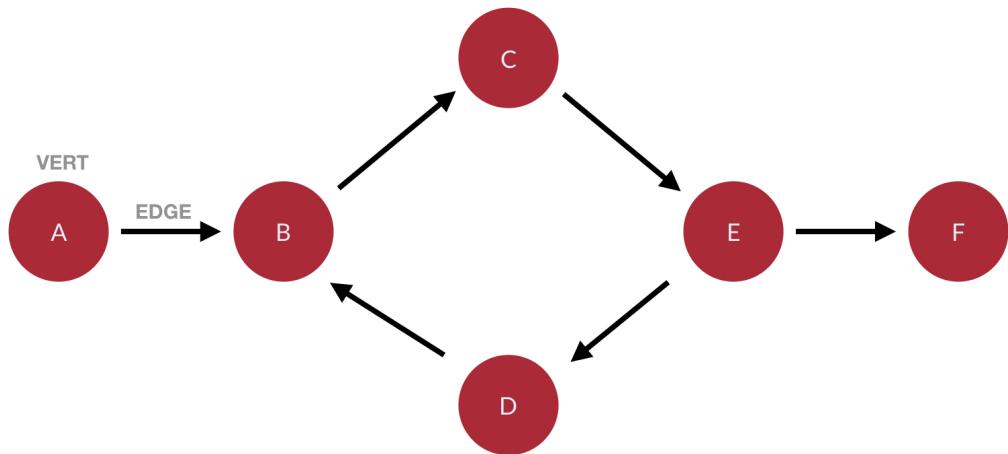
Objective 01 - Describe what a graph is, explain its components, provide examples of its useful applications, and draw each of the different graph types

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



<https://camo.githubusercontent.com/134d2271d2ff5cedb2f05ede63326cb12f8253d2/68747470733a2f2f692e696d6775722e636f6d2f456232536b68482e6a7067>

We call the nodes in a graph **vertices** (or **vertices** or **verts**), and we call the connections between the verts **edges**.

An edge denotes a relationship or linkage between the two vertices.

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.

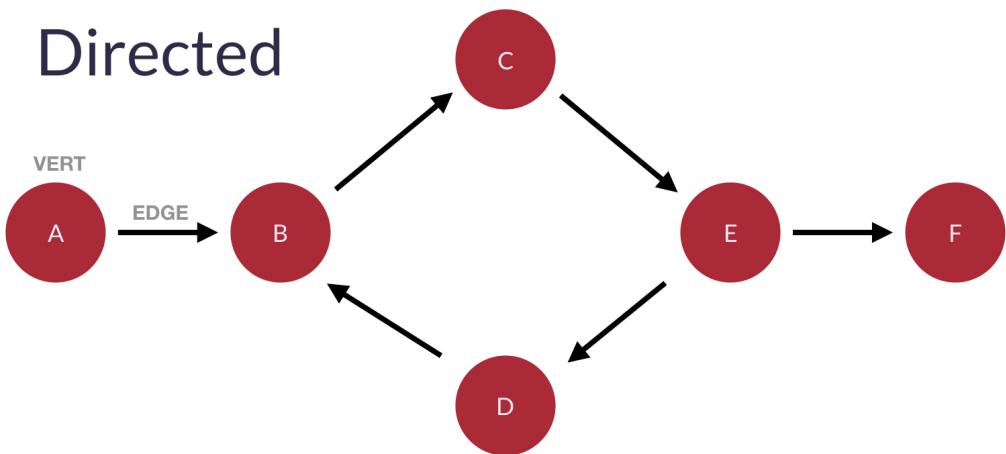
It could represent the money owed in an ongoing poker night amongst friends.

And so on.

Types of Graphs

Directed and Undirected Graphs

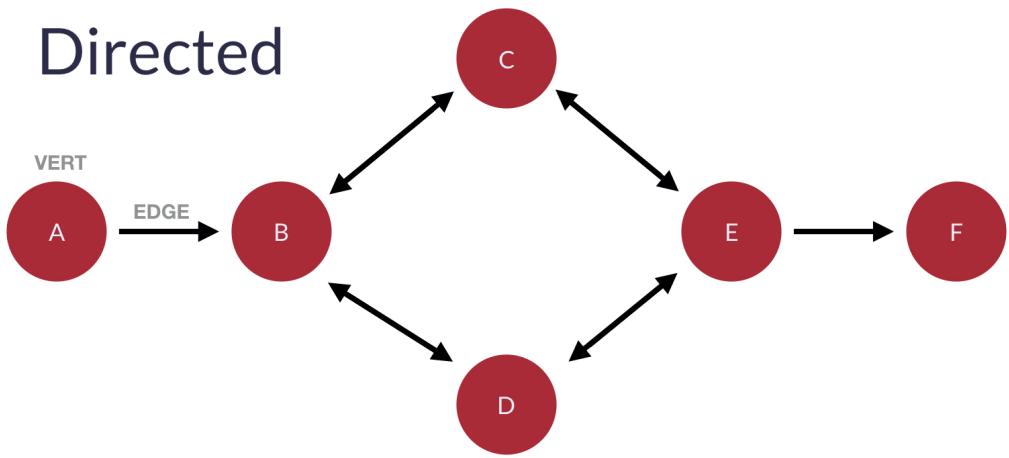
The nature of the relationship that we represent determines if we should use a directed or undirected graph. If we could describe the relationship as "one way", then a directed graph makes the most sense. For example, representing the owing of money to others (debt) with a directed graph would make sense.



<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.

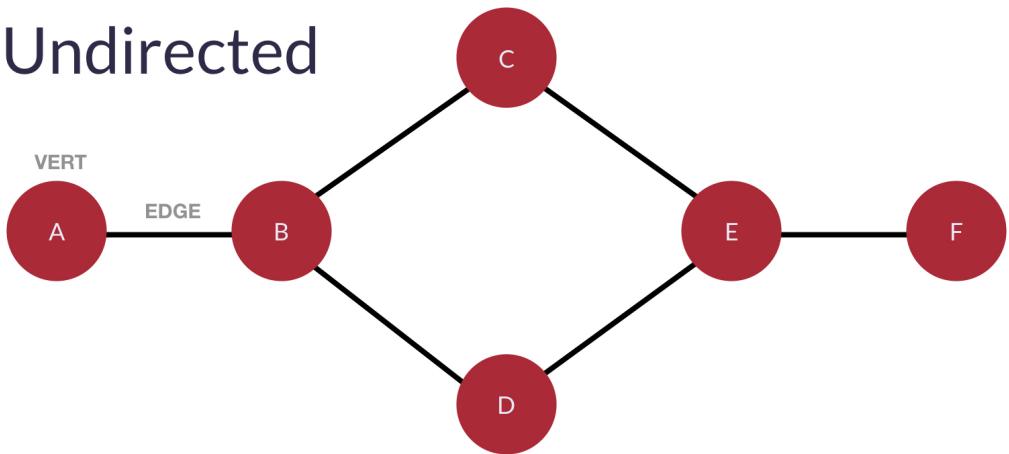
Directed



<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.

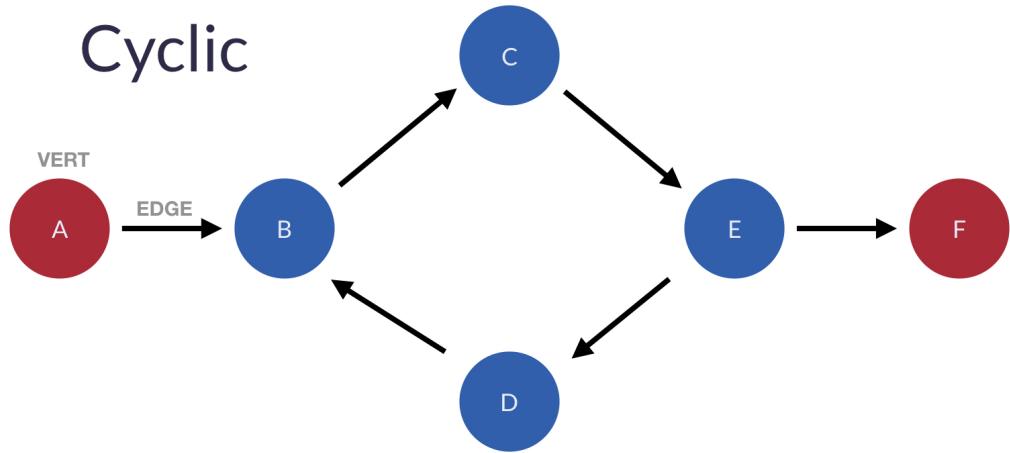
Undirected



<https://camo.githubusercontent.com/6b782a86e1920d53411b88e1510c7efd0bed2bc2/68747470733a2f2f692e696d6775722e636f6df534a4e3036776a2e6a7067>

Cyclic and Acyclic Graphs

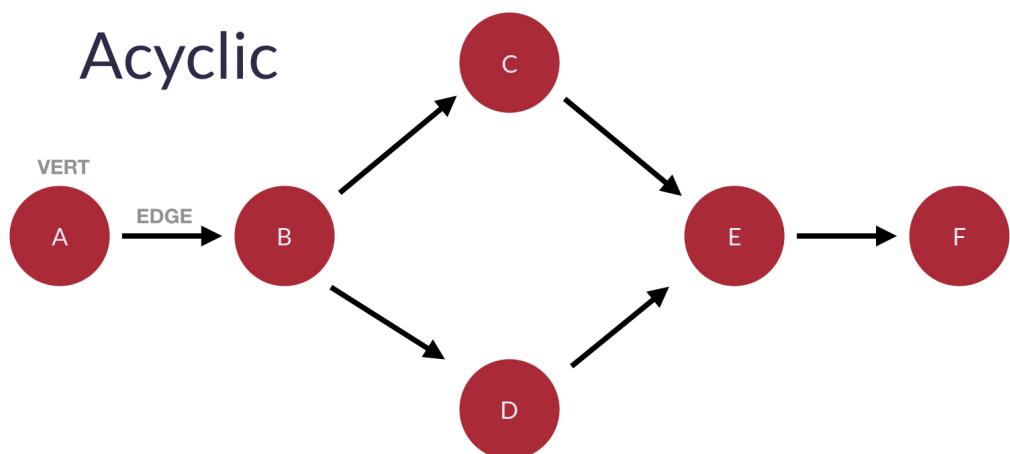
If you can form a cycle (for example, follow the edges and arrive again at an already-visited vert), the graph is **cyclic**. For instance, in the image below, you can start at B and then follow the edges to C, E, D, and then back to B (which you've already visited).



<https://camo.githubusercontent.com/e23529f1bd2dfe3227dee64fe174252b0c310d1a/68747470733a2f2f692e696d6775722e636f6d2f58764d44616c302e6a7067>

Note: any undirected graph is automatically cyclic since you can always travel back across the same edge.

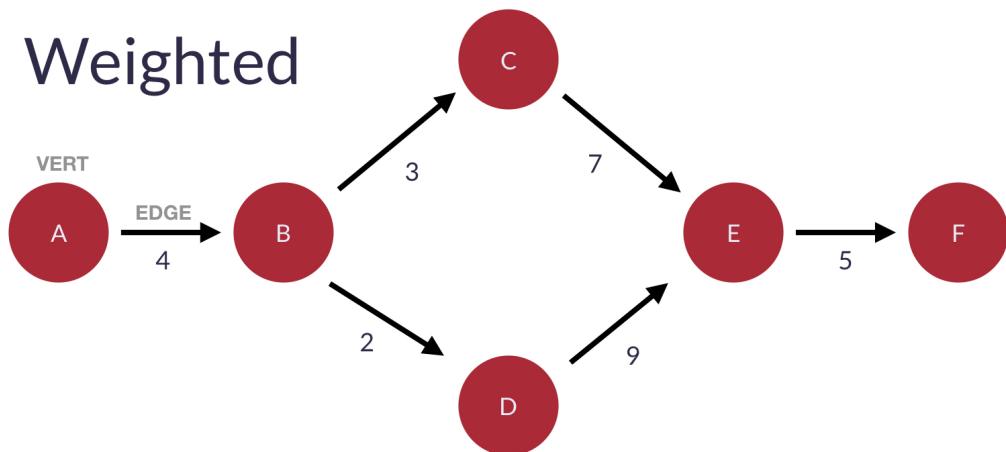
If you cannot form a cycle (for example, you cannot arrive at an already-visited vert by following the edges), we call the graph **acyclic**. In the example below, no matter which vert you start at, you cannot follow edges in such a way that you can arrive at an already-visited vert.



<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**.



<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.

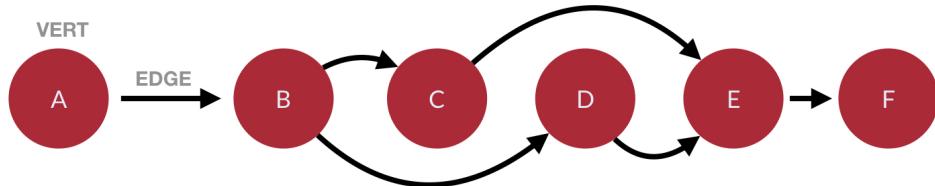
We can further modify weights. For example, if you were building a graph representing a map for bicycle routes, we could give roads with bad car traffic or very steep inclines unnaturally large weights. That way, a routing algorithm would be unlikely to take them. (This is how Google Maps avoids freeways 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.

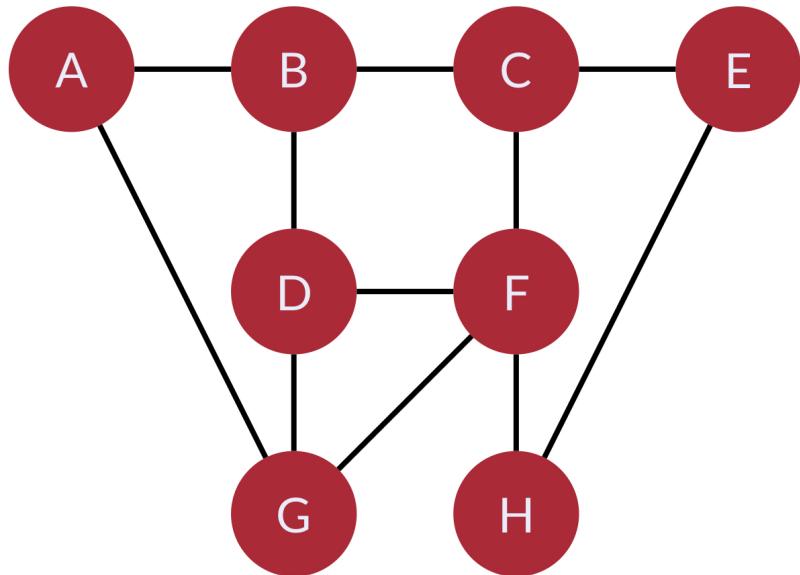
Follow Along

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 vertices and then connect them with solid lines (not arrows) anyway we see fit.

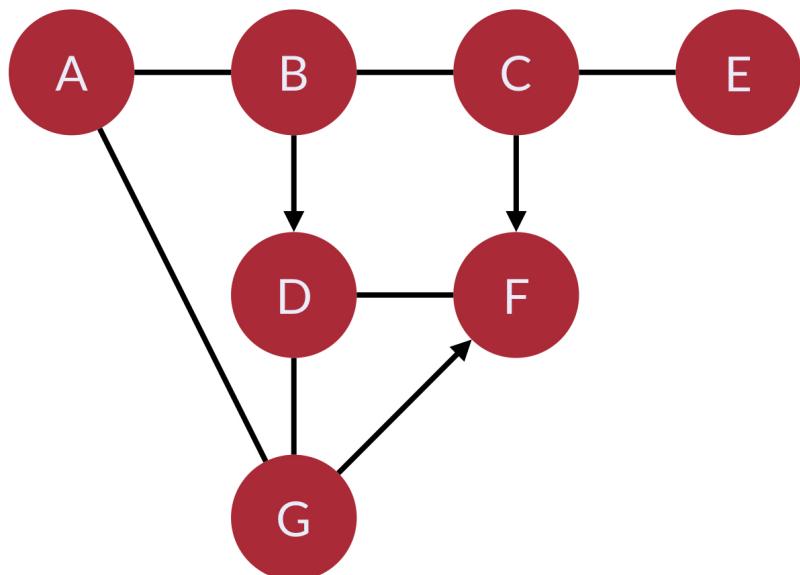


<https://camo.githubusercontent.com/de48aaaa53f7fead48f3d09e394f8b1342f7d226/68747470733a2f2f692e696d6775722e636f6d2f6d6964443358642e6a7067>

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.



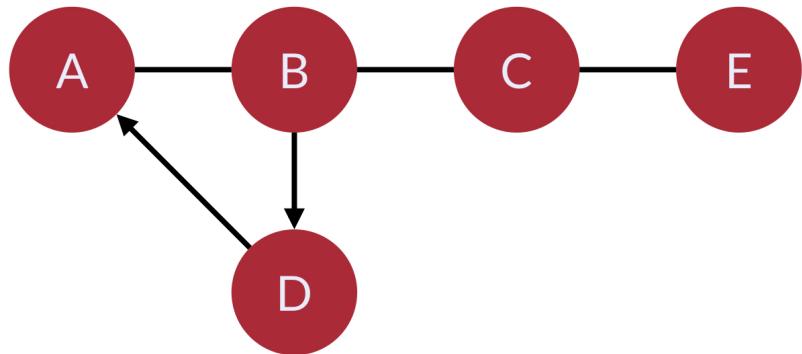
<https://camo.githubusercontent.com/53c2b80679e6731818f5bb72fecdf17579dd0f70/68747470733a2f2f692e696d6775722e636f6d2f3870436f6568412e6a7067>

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.



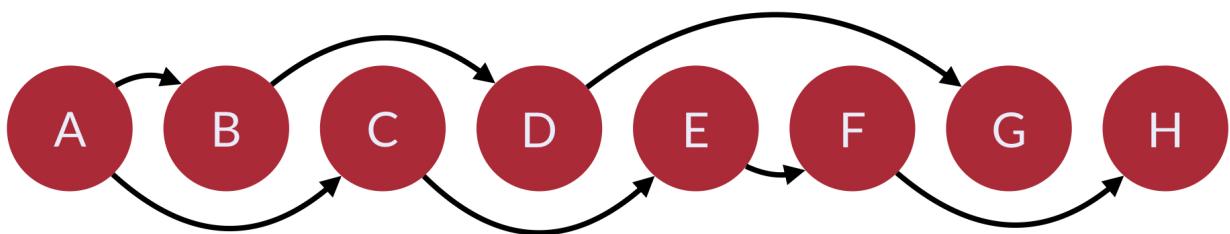
<https://camo.githubusercontent.com/8609287c7507ded66414ab2be7154d68436b82ac/68747470733a2f2f692e696d6775722e636f6d2f4a424f7572506e2e6a7067>

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> (Links to an external site.)



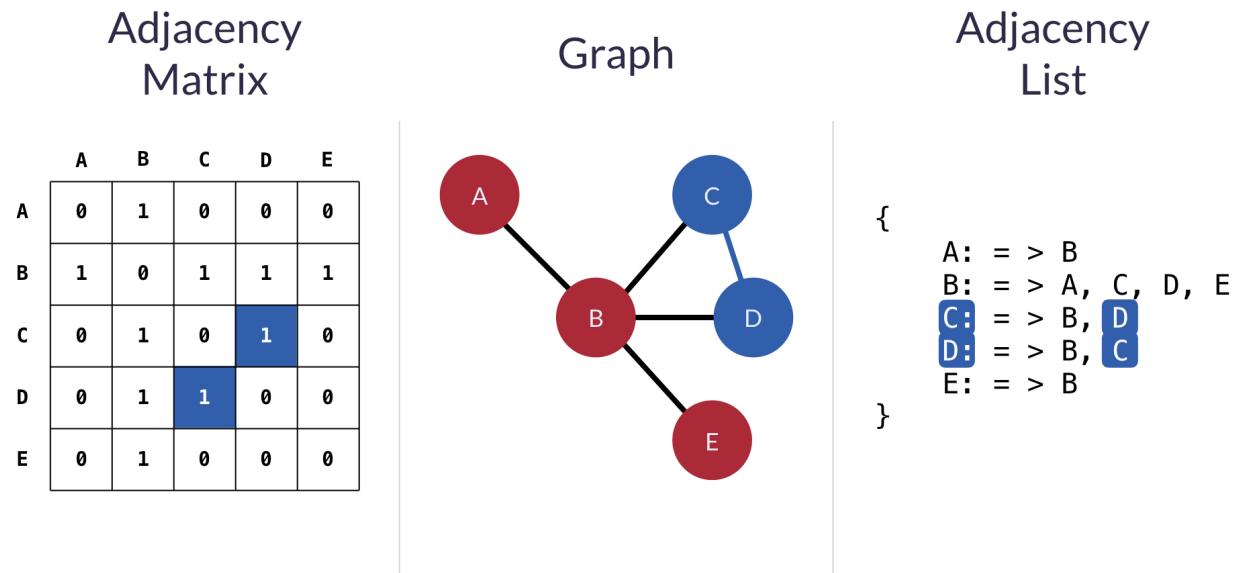
Objective 02 - Represent a graph as an adjacency list and an adjacency matrix and compare and contrast the respective representations

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.

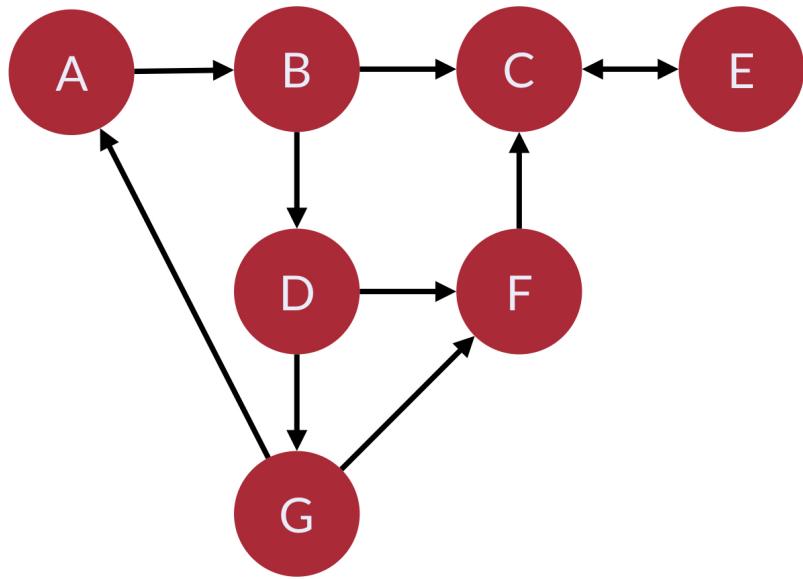
Below is an example of how we would represent a graph with an adjacency matrix and an adjacency list. Notice how we represent the relationship between verts C and D when using each type.



<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:

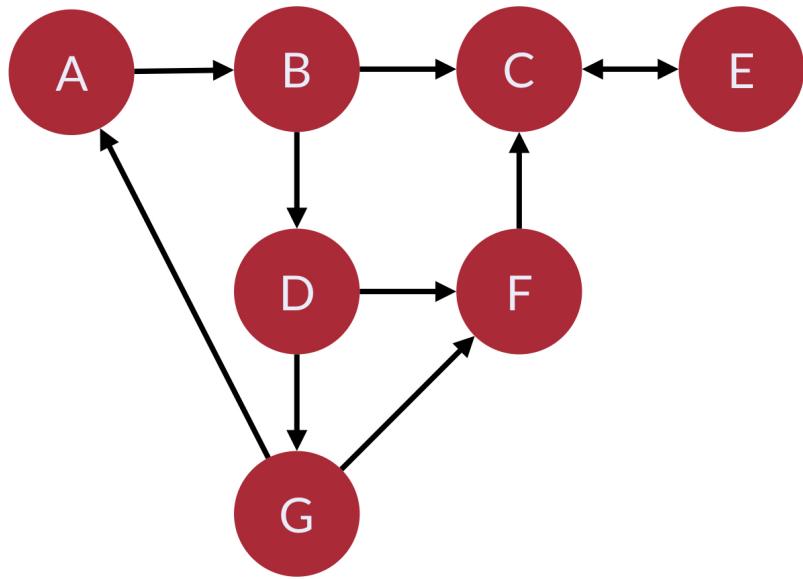
```

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.

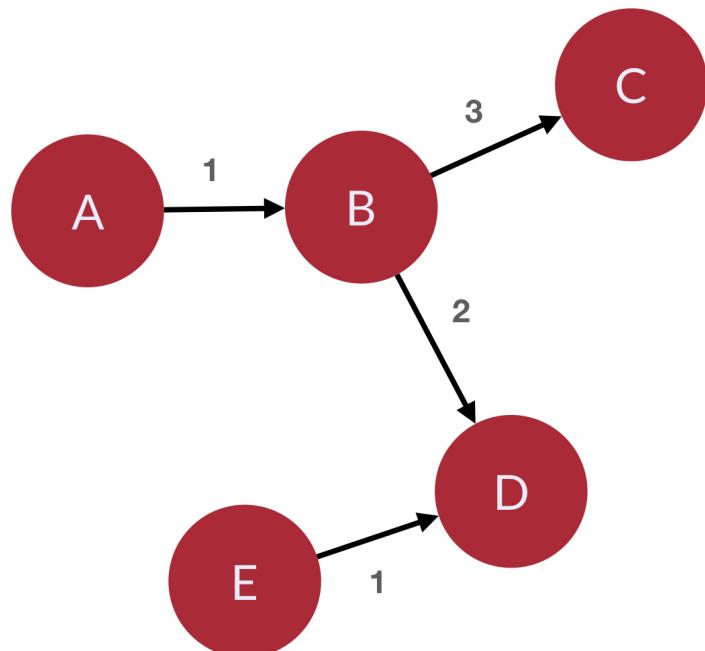
Summary

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    }
```

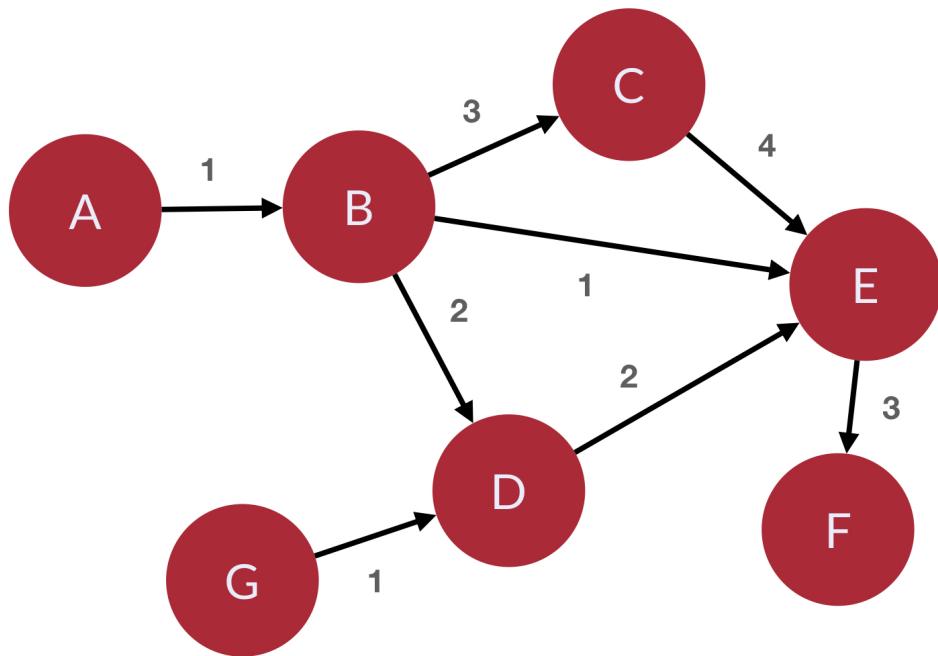
The difference between this implementation and the previous adjacency list is that this representation allows our edges to have weights.

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 self.connections])
```

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()
```

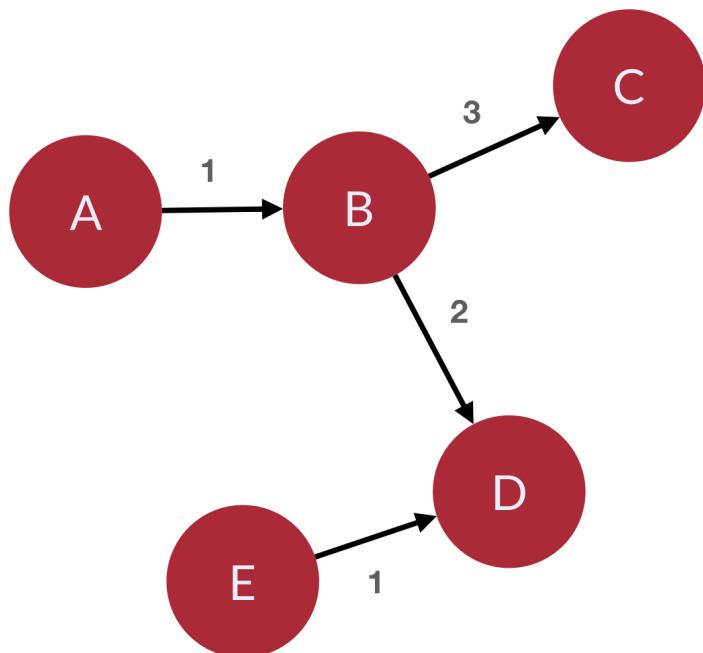
Let's go ahead and test our class definitions and build up a graph structure in a Python interactive environment.

```
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 0x7fd0f183fdc0>, 2: <__main__.Vertex
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/>



D2-Graphs 2

CS46 Graphs II.ipynb

<https://gist.github.com/bgoonz/4dc35438f8c293cf68e81c0d73ddfe1a>



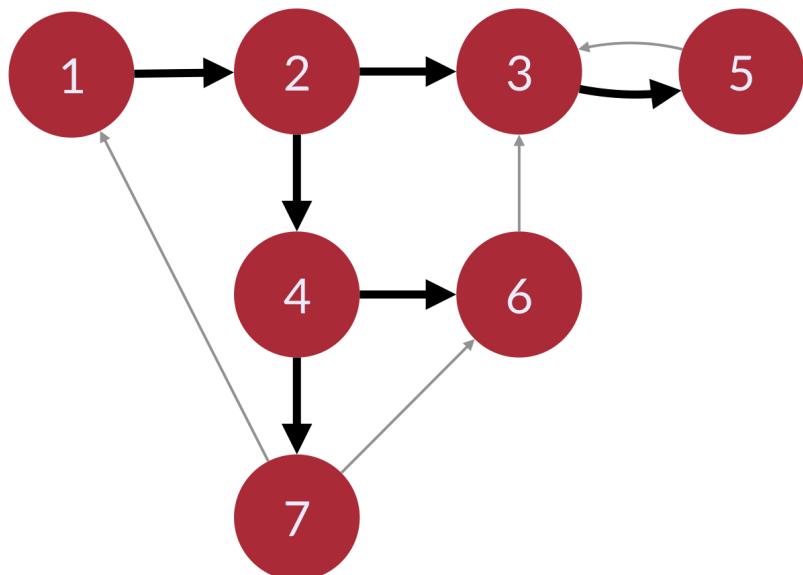
Objective 01 - Represent a breadth-first-search of a graph in pseudo-code and recall typical applications for its use

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 visitation order in a BFS:



<https://camo.githubusercontent.com/b22308d5dd2fe7ee7f295f89482bdab2f8e8976f/68747470733a2f2f692e696d6775722e636f6d2f314c506e4f41582e6a7067>

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 of 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": {"H"},  
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        }  
19
```

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/>



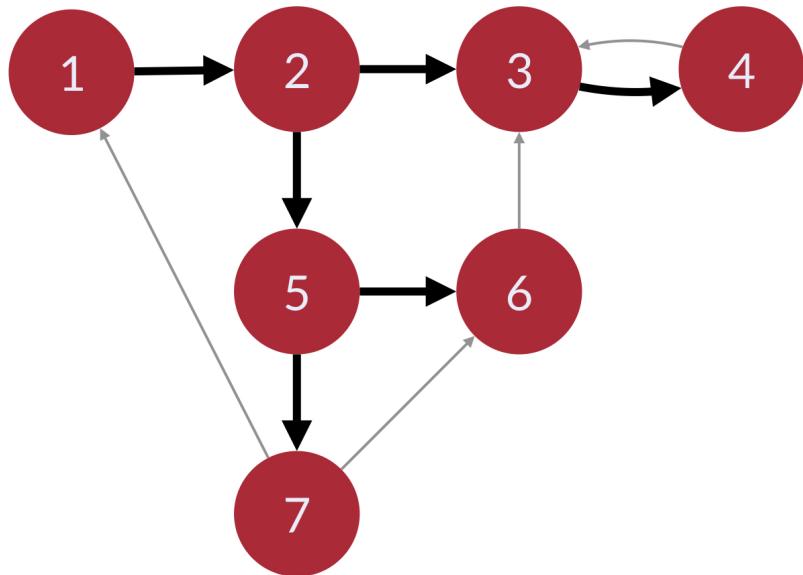
Objective 02 - Represent a depth-first-search of a graph in pseudo-code and recall typical applications for its use

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:



<https://camo.githubusercontent.com/30ab40a62286d6fe39210efa52f5b802f383daa0/68747470733a2f2f692e696d6775722e636f6d2f565654433959782e6a7067>

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/> (Links to an external site.)



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
6     def __contains__(self, vert):
7         return vert in self.vertices
8
9     def __iter__(self):
10        return iter(self.vertices.values())
```

```

11     def add_vertex(self, value):
12         self.count += 1
13         new_vert = Vertex(value)
14         self.vertices[value] = new_vert
15         return new_vert
16
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/> (Links to an external site.)



Objective 04 - Implement a depth-first search on a graph

Overview

The depth-first search algorithm on a graph starts at an arbitrary vertex in the graph and explores as far as possible down each branch before backtracking. So, you start at the starting vertex, mark it as visited, and then move to an adjacent unvisited vertex. You continue this loop until every reachable vertex is visited.

Follow Along

Before defining our depth-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 `depth_first_search` method to our `Graph` class. One of the most common and simplest ways to implement a DFS is to use a set to keep track of visited vertices and use recursion to manage the visitation order. Let's now define our function:

```

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()
27
28     def depth_first_search(self, vertex, visited = set()):
29         visited.add(vertex)
30         for next_vert in vertex.get_connections():
31             if next_vert not in visited:
32                 self.depth_first_search(next_vert, visited)
```

Challenge

1. Does a depth-first search reliably find the shortest path?
2. If you didn't want to use recursion, what data structure could you use to write an iterative depth-first search?

Additional Resources

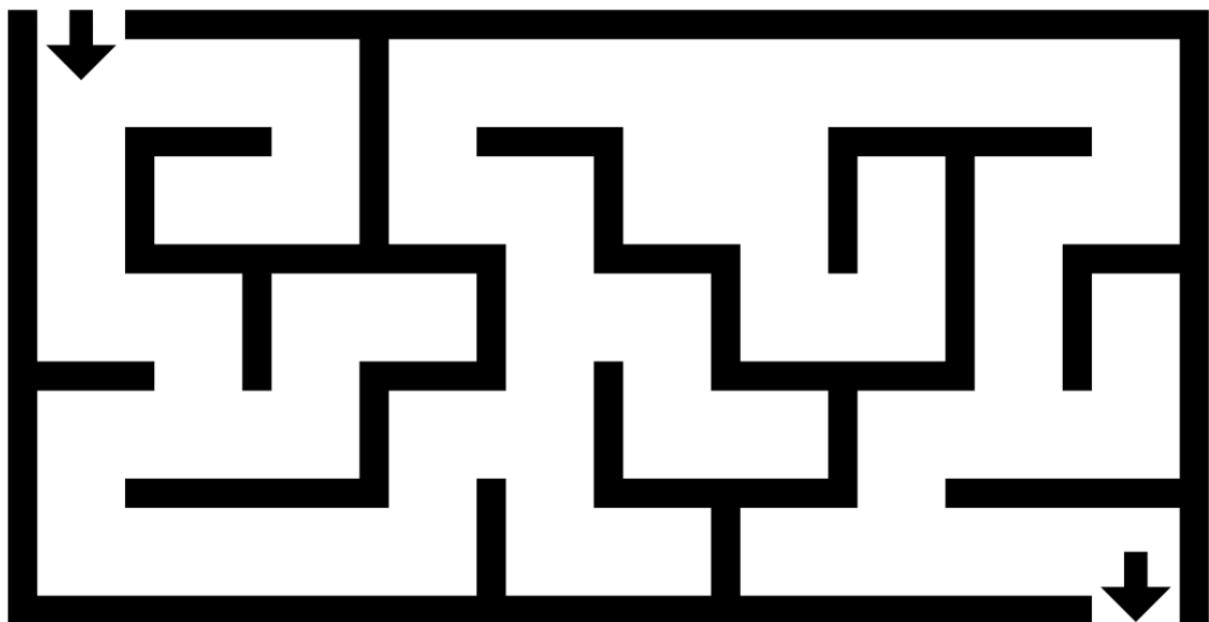
- <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/> (Links to an external site.)



DFS

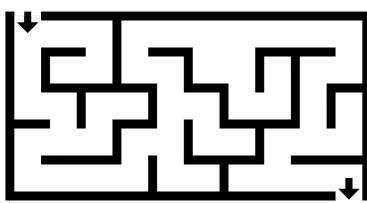
Depth-first search (DFS) is an [algorithm](#) for searching a [graph](#) or [tree](#) data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored. Many problems in computer science can be thought of in terms of graphs. For example, analyzing networks, mapping routes, scheduling, and finding [spanning trees](#) are graph problems. To analyze these problems, [graph-search algorithms](#) like depth-first search are useful.

Depth-first searches are often used as subroutines in other more complex algorithms. For example, the [matching algorithm](#), [Hopcroft-Karp](#), uses a DFS as part of its algorithm to help to find a [matching](#) in a graph. DFS is also used in [tree-traversal](#) algorithms, also known as tree searches, which have applications in the [traveling-salesman problem](#) and the [Ford-Fulkerson algorithm](#).



How do you solve a maze?

Depth-first search is a common way that many people naturally approach solving problems like mazes. First, we select a path in the maze (for the sake of the example, let's choose a path according to some rule we lay out ahead of time) and we follow it until we hit a dead end or reach the finishing point of the maze. If a given path doesn't work, we backtrack and take an alternative path from a past junction, and try that path. Below is an animation of a DFS approach to solving this maze.



DFS is a great way to solve mazes and other puzzles that have a single solution.

Contents

- [Depth-first Search](#)
- [Implementing Depth-first Search](#)
- [Complexity of Depth-first Search](#)
- [Applications](#)
- [References](#)

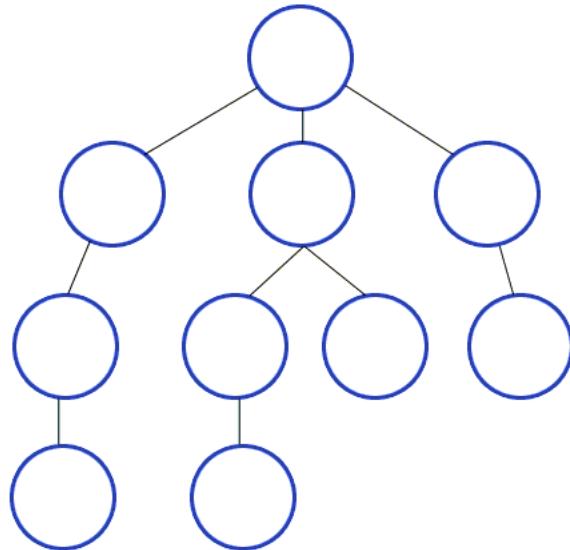
Depth-first Search

The main strategy of depth-first search is to explore deeper into the graph whenever possible. Depth-first search explores edges that come out of the most recently discovered vertex, s_s . Only edges going to unexplored vertices are explored. When all of s_s 's edges have been explored, the search backtracks until it reaches an unexplored neighbor. This process continues until all of the vertices that are reachable from the original source vertex are discovered. If there are any unvisited vertices, depth-first search selects one of them as a new source and repeats the search from that vertex. The algorithm repeats this entire process until it has discovered every vertex. This algorithm is careful not to repeat vertices, so each vertex is explored once. DFS uses a [stack](#) data structure to keep track of vertices.

Here are the basic steps for performing a depth-first search:

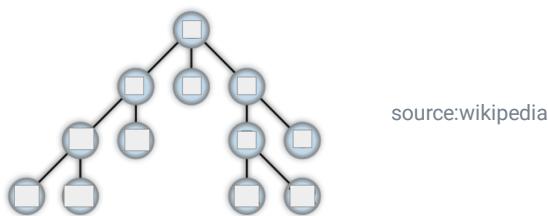
- Visit a vertex s_s .
- Mark s_s as visited.
- Recursively visit each unvisited vertex attached to s_s .

This animation illustrates the depth-first search algorithm:



Note: This animation does not show the marking of a node as "visited," which would more clearly illustrate the backtracking step.

Fill out the following graph by labeling each node 1 through 12 according to the order in which the depth-first search would visit the nodes:



Show solution

Implementing Depth-first Search

Below are examples of pseudocode and Python code implementing DFS both recursively and non-recursively. This algorithm generally uses a [stack](#) in order to keep track of visited nodes, as the last node seen is the next one to be visited and the rest are stored to be visited later.

Pseudocode[1]

Python Implementation without Recursion

DFS can also be implemented using recursion, which greatly reduces the number of lines of code.

Python Implementation Using Recursion

It is common to modify the algorithm in order to keep track of the edges instead of the vertices, as each edge describes the nodes at each end. This is useful when one is attempting to reconstruct the traversed tree after processing each node. In case of a forest or a group of trees, this algorithm can be expanded to include an outer loop that iterates over all trees in order to process every single node.

There are three different strategies for implementing DFS: *pre-order*, *in-order*, and *post-order*.

Pre-order DFS works by visiting the current node and successively moving to the left until a leaf is reached, visiting each node on the way there. Once there are no more children on the left of a node, the children on the right are visited. This is the most standard DFS algorithm.

Instead of visiting each node as it traverses down a tree, an **in-order** algorithm finds the leftmost node in the tree, visits that node, and subsequently visits the parent of that node. It then goes to the child on the right and finds the next leftmost node in the tree to visit.

A **post-order** strategy works by visiting the leftmost leaf in the tree, then going up to the parent and down the second leftmost leaf in the same branch, and so on until the parent is the last node to be visited within a branch. This type of algorithm prioritizes the processing of leaves before roots in case a goal lies at the end of a tree.

Complexity of Depth-first Search

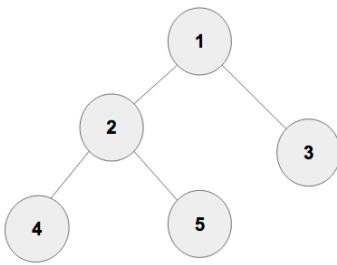
Depth-first search visits every vertex once and checks every edge in the graph once. Therefore, DFS complexity is $O(V + E)O(V+E)$. This assumes that the graph is represented as an [adjacency list](#).

DFS vs BFS

[Breadth-first search](#) is less space-efficient than depth-first search because BFS keeps a priority queue of the entire frontier while DFS maintains a few pointers at each level.

If it is known that an answer will likely be found far into a tree, DFS is a better option than BFS. BFS is good to use when the depth of the tree can vary or if a single answer is needed—for example, the shortest path in a tree. If the entire tree should be traversed, DFS is a better option.

BFS always returns an optimal answer, but this is not guaranteed for DFS.



Here is an example that compares the order that the graph is searched in when using a BFS and then a DFS (by each of the three approaches).[\[2\]](#)

Breadth First Search : 1 2 3 4 5

Depth First Search

- Pre-order: 1 2 4 5 3
- In-order : 4 2 5 1 3
- Post-order : 4 5 2 3 1

Applications

Depth-first search is used in [topological sorting](#), [scheduling problems](#), [cycle detection](#) in graphs, and solving puzzles with only one solution, such as a maze or a [sudoku](#) puzzle.

Other applications involve analyzing networks, for example, testing if a graph is [bipartite](#). Depth-first search is often used as a subroutine in [network flow](#) algorithms such as the [Ford-Fulkerson algorithm](#).

DFS is also used as a subroutine in [matching algorithms](#) in [graph theory](#) such as the [Hopcroft-Karp algorithm](#).

Depth-first searches are used in mapping routes, scheduling, and finding [spanning trees](#).

D4



Utilities

Code lab Notebooks

 [part1.ipynb](#)

<https://gist.github.com/bgoonz/f2f3e9606771b2862a88b0694dba9858#file-part1-ipynb>

 [part3.ipynb](#)

<https://gist.github.com/bgoonz/b72af37df7ac1e67c623c1205263dce0#file-part3-ipynb>

 [part2.ipynb](#)

<https://gist.github.com/bgoonz/f0980ef8b66987b971823eae3ee1576#file-part2-ipynb>

 [Number Bases and Chars.ipynb](#)

<https://gist.github.com/bgoonz/85cf385ba5382cea548c2b6083cd1b3f#file-number-bases-and-chars-ipynb>

 [Copy of ArraysAndStrings.ipynb](#)

<https://gist.github.com/bgoonz/6161262b9e48fc622782a85bbb804852#file-copy-of-arraysandstrings-ipynb>

 [CS47 Python III.ipynb](#)

<https://gist.github.com/bgoonz/8a92c0f02d7847eb67e3732384d4d03b#file-cs47-python-iii-ipynb>

 [part4.ipynb](#)

<https://gist.github.com/bgoonz/40da977a7998915937f8fd6d53e5072a#file-part4-ipynb>

 [CS47 Python II.ipynb](#)

<https://gist.github.com/bgoonz/4323443e5b3b4eb8b4b880e17b0612c4#file-cs47-python-ii-ipynb>

 [Copy of Searching.ipynb](#)

<https://gist.github.com/bgoonz/d4cf92a32fc64bfa4435107fa6a699#file-copy-of-searching-ipynb>

 [Copy of qandstack.ipynb](#)

<https://gist.github.com/bgoonz/672f6599cc5bf68e68dd34a5acb2fe13#file-copy-of-qandstack-ipynb>

 HT2.ipynb

<https://gist.github.com/bgoonz/afb19b48df5cddd69a9fe95e1b1b4495#file-ht2-ipynb>

 BST.ipynb

<https://gist.github.com/bgoonz/c6ddd80d7ae24eca670f5fbebb7795f0>

 Copy of CS46 Graphs II.ipynb

<https://gist.github.com/bgoonz/0236f68e179be2cf39774531364645f8#file-copy-of-cs46-graphs-ii-ipynb>

Repl.IT

 week-10-take-5

<https://replit.com/@bgoonz/week-10-take-5#2-resources/general/cheat-sheets/0-OFFICIALLY-ALLOWED-Sequelize-Cheat-sheet.md>

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

Trinket

1. Numbers and math
2. Logic
3. Words and letters
4. Changing text
5. Variable containers
6. Conditionals
7. Lists
8. Loops
9. Dictionaries



iframe inception (forked) - CodeSandbox

[https://codesandbox.io/s/iframe-inception-forked-j5ofi?
file=/index.html](https://codesandbox.io/s/iframe-inception-forked-j5ofi?file=/index.html)

Numbers in Python



Put Python Anywhere on the Web

<https://trinket.io/python3/61c47663ad>



Getting Started with Python

<https://docs.trinket.io/getting-started-with-python#/numbers/order-of-operations>



Learn Python, Part 1: Numbers

<https://learnpython.trinket.io/learn-python-part-1-numbers#/welcome/where-we'll-go>



Python Challenges

<https://hourofpython.trinket.io/python-challenges#/string-challenges/number-of-things-challenge>

Learn Python, Part 8: Loops



Learn Python, Part 8: Loops

<https://learnpython.trinket.io/learn-python-part-8-loops#/welcome/where-well-go>



Utilities

Search Awesome Resources:

 **Awesome Search**

<https://search-awesome.vercel.app/>

 **Search Awesome - CodeSandbox**

<https://codesandbox.io/s/search-awesome-iomg4>

 **3.9.7 Documentation**

<https://docs.python.org/3/>

 **ds-algo (forked) - CodeSandbox**

<https://codesandbox.io/s/ds-algo-forked-60s1b>

Number Base Converter:

 **React App**

<http://number-base-converter-react.vercel.app/>

 **ds-algo (forked) - CodeSandbox**

<https://codesandbox.io/s/ds-algo-forked-lfujh?from-embed>

 **DS-Algo-Codebase**

<https://bgoonz-branch-the-algos.vercel.app/>



TheALGOS - CodeSandbox

<https://codesandbox.io/s/thealgos-q48t0>



Python Notes

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



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ds-algo-forked-lfujh?from-embed>

YouTube

 CS47 Intro to Python I w/ Tom Tarpey

<https://youtu.be/ocxkkzjdFeY>

 CS47 Python II w/ Tom Tarpey

<https://youtu.be/j5Hu08FhAJQ>

 toms_prev_lectures

<https://www.youtube.com/playlist?list=PLWX9jswdDQ0UILD-9oZhBCGcPYWFdrMQA>

 CS47 Python III w/ Tom Tarpey

<https://youtu.be/rxYrTtxefjE>

 CS47 Strings and Arrays Python IV w/ Tom Tarpey

<https://youtu.be/AKDIKZ6zwmw>

 CS47 Hash Tables I w/ Tom Tarpey

<https://youtu.be/raIVrpEE3vs>



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>

 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

 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>

Code Lab Notebook Embeds From Lecture

practice

Supplemental Practice:

 PYTHON_PRAC-1

<https://replit.com/@bgoonz/PYTHONPRAC-1>

 py-prac-medium-2

<https://replit.com/@bgoonz/py-prac-medium-2#prac13.py>

ADS Implementations:

→ [Array](#)

/practice/untitled/array

→ [Home](#)

/

→ [Queue](#)

/data-structures/untitled/queue

→ [Binary Search](#)

/data-structures/untitled/binary-search

→ [Binary Tree](#)

/practice/untitled/binary-tree

→ [Binary Search Tree](#)

/data-structures/untitled/binary-search-tree

→ [Recursion](#)

/practice/untitled/untitled-6

→ [Hash Table](#)

/practice/untitled/untitled-5

→ [Linked List](#)

/data-structures/untitled/linked-list

→ [Sorting](#)

/practice/untitled/untitled-3

→ [ADS Implementations:](#)

/practice/untitled

→ [ADS Implementations:](#)

/practice/untitled

→ [ADS Implementations:](#)

/practice/untitled

→ [ADS Implementations:](#)

/practice/untitled

Industry Standard Algorithms

 main-prac

https://replit.com/@bgoonz/main-prac#directed_graph.py

Implement a function recursively to get the desired Fibonacci sequence value. Your code should have the same input/output as the iterative code in the instructions.

```
1 def get_fib(position):
2     output = 0
3     if(position==0):
4         return output
5
6     if(position==1):
7         return position
8     else:
9         output += get_fib(position-1)+get_fib(position-2)
10    return output
11
12
13 # Test cases
14 print get_fib(9)
15 print get_fib(11)
16 print get_fib(0)
17
```

Interview Practice Resources

Guides

- Interview Cake Algs/DS reference
- Algorithms at GeeksforGeeks

Live Interviewing

- Pramp: live interview practice with peers

Problem lists

- Techie Delight, with solutions

Code Challenge Sites

Easy to Medium

- Coderbyte
- Codewars
- CodeFights

Medium to Difficult

- TopCoder
- HackerRank
- LeetCode
- GeeksforGeeks

Overflow Practice Problems

- 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): ($1 \leq n \leq 2 \times 10^9$).

[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 ($1 \leq n \leq 10,000,000$).

[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 ($0 \leq n \leq 100$)

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 ($0 \leq a,b,c,d \leq 9$) where $(a + b + c + d)$ will be equal to n . [Go to the editor](#)

Input:

n ($1 \leq n \leq 50$)

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° angle, and the latter sometimes refers specifically to a rhombus with a 45° 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:

$n (n \leq 10000)$. 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(xp,yp), Q(xq,yq), R(xr,yr) and S(xs,ys). Write a Python program to test AB and CD are orthogonal or not. [Go to the editor](#)

Input:

xp,yp, xq, yq, xr, yr, xs and ys 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 YYYYYY, 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

XYZZZZZZ1000023

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
```

x_i, y_i 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;

1.5000000.

[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 ≤ 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", "apple", "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:

w3resuorce

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", "ypth"]

["python", "ypths"]

["python", "ython"]

["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:

0

(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

$1634 = 1^4 + 6^4 + 3^4 + 4^4$

$8208 = 8^4 + 2^4 + 0^4 + 8^4$

$9474 = 9^4 + 4^4 + 7^4 + 4^4$

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:

```
("PPYYYYTHON")  
("PPyyythonnn")  
("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,

122333444455556666677777788888889999999990 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><a href="[[[https://www.iana.org/domains/example">More](https://www.iana.org/domains/example">More]
(https://www.iana.org/domains/example">More)%28https://www.iana.org/domains/example">More]
(https://www.iana.org/domains/example">More)%28https://www.iana.org/domains/example">More)%28https://www.iana.org/domains/
example">More)%28https://www.iana.org/domains/example">More%29)\\" information...</a></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: PHP

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: PHP

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: JvScrpt

[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 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 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’, ‘0.55’, ‘0.55’, ‘0.55’, ‘0.54’, ‘0.54’, ‘0.55’, ‘0.54’, ‘0.55’, ‘0.54’]

List of Floats:

[0.49, 0.54, 0.54, 0.54, 0.54, 0.55, 0.54, 0.54, 0.54, 0.55, 0.55, 0.55, 0.54, 0.55, 0.54, 0.55, 0.54, 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: 00011100011

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](#)

Array

```
1 # Python Program to demonstrate creation of Array using array creations
2 import array as arr
3
4 print("INTEGER ARRAY OPERATIONS:")
5 size = int(input(" Enter the size of Array: "))
6 # creating an array with integer type
7 lst = list()
8 for i in range(size):
9     lst.append(int(input("Enter the Integer Element:")))
10 n = arr.array(lst)
11
12 # printing array
13 def pr(n):
14     print("The new integer array is : ", end=" ")
15     for i in range(len(n)):
16         print(n[i], end =", ")
17     print()
18
19 def add(n,j):
20     print("The Array before adding: ", end=" ")
21     for i in range(len(n)):
22         print(n[i], end =", ")
23     print()
24     #Append() method
25     n.append(j)
26     print("The Array After adding: ", end=" ")
27     for i in range(len(n)):
28         print(n[i], end =", ")
29     print()
30
31 def adde(n,j,p):
32     print("The Array before adding: ", end=" ")
33     for i in range(len(n)):
34         print(n[i], end =", ")
35     print()
36     #insert() method
37     n.insert(p,j)
38     print("The Array After adding: ", end=" ")
39     for i in range(len(n)):
40         print(n[i], end =", ")
41     print()
42
43 def pp(n,j):
44     if n:
45         print("The Array before Popping: ", end=" ")
46         for i in range(len(n)):
47             print(n[i], end =", ")
48         print()
49         # pop() method
50         n.pop(j)
51         print("The Array After Popping: ", end=" ")
52         for i in range(len(n)):
53             print(n[i], end =", ")
54         print()
55     else:
56         print("Array Empty")
57
58 def prt(n,j):
59     if n:
60         if j in range(len(n)):
61             print("The Array before Removing: ", end=" ")
62             for i in range(len(n)):
63                 print(n[i], end =", ")
64             print()
65             #remove Method
66             n.remove(j)
67             print("The Array After Removing: ", end=" ")
68             for i in range(len(n)):
69                 print(n[i], end =", ")
70             print()
71         else:
```

```
72         print("Index Out of Bound")
73     else:
74         print("Array Empty")
75
76 #Driver code
77 flag = 1
78 while(flag):
79     print()
80     print("1.Print Array\n2.Add Element using append()\n3.Add Element using insert()\n4.Pop() Element\n5.Remove El")
81     option = int(input("Enter the option :"))
82     if option == 1:
83         pr(n)
84     elif option == 2:
85         i = int(input("Enter the Element to be added: "))
86         add(n,i)
87     elif option == 3:
88         p = int(input("Enter the position to add element:"))
89         i = int(input("Enter the Element: "))
90         adde(n, i, p)
91     elif option == 4:
92         i = int(input("Enter the Index position To be popped: "))
93         pp(n, i)
94     elif option == 5:
95         i = int(input("Enter the Element Position To be Removed: "))
96         prt(n, i)
97     elif option == 6 :
98         flag = 0
99
100
101
102
103
104
105
```


Stack

Queue

Implementation

```
1 """Make a Queue class using a list!
2 Hint: You can use any Python list method
3 you'd like! Try to write each one in as
4 few lines as possible.
5 Make sure you pass the test cases too!"""
6
7 class Queue:
8     def __init__(self, head=None):
9         self.storage = [head]
10
11     def enqueue(self, new_element):
12         if(self.storage):
13             self.storage.append(new_element)
14         else:
15             self.storage = [new_element]
16         return new_element
17
18     def peek(self):
19         if(self.storage):
20             return self.storage[0]
21         else:
22             return None
23
24     def dequeue(self):
25         if(self.storage):
26             return self.storage.pop(0)
27         else:
28             return None
29
30 # Setup
31 q = Queue(1)
32 q.enqueue(2)
33 q.enqueue(3)
34
35 # Test peek
36 # Should be 1
37 print(q.peek())
38
39 # Test dequeue
40 # Should be 1
41 print(q.dequeue())
42
43 # Test enqueue
44 q.enqueue(4)
45 # Should be 2
46 print(q.dequeue())
47 # Should be 3
48 print(q.dequeue())
49 # Should be 4
50 print(q.dequeue())
51 q.enqueue(5)
52 # Should be 5
53 print(q.peek())
54
```

Second Tab

```
1 """Make a Queue class using a list!
2 Hint: You can use any Python list method
3 you'd like! Try to write each one in as
4 few lines as possible.
5 Make sure you pass the test cases too!"""
6
7 class Queue:
8     def __init__(self, head=None):
9         self.storage = [head]
10
11    def enqueue(self, new_element):
12        if(self.storage):
13            self.storage.append(new_element)
14        else:
15            self.storage = [new_element]
16        return new_element
17
18    def peek(self):
19        if(self.storage):
20            return self.storage[0]
21        else:
22            return None
23
24    def dequeue(self):
25        if(self.storage):
26            return self.storage.pop(0)
27        else:
28            return None
29
30 # Setup
31 q = Queue(1)
32 q.enqueue(2)
33 q.enqueue(3)
34
35 # Test peek
36 # Should be 1
37 print(q.peek())
38
39 # Test dequeue
40 # Should be 1
41 print(q.dequeue())
42
43 # Test enqueue
44 q.enqueue(4)
45 # Should be 2
46 print(q.dequeue())
47 # Should be 3
48 print(q.dequeue())
49 # Should be 4
50 print(q.dequeue())
51 q.enqueue(5)
52 # Should be 5
53 print(q.peek())
54
```

Binary Search

```
1 # Uses python3
2 import random
3 """You're going to write a binary search function.
4 You should use an iterative approach - meaning
5 using loops.
6 Your function should take two inputs:
7 a Python list to search through, and the value
8 you're searching for.
9 Assume the list only has distinct elements,
10 meaning there are no repeated values, and
11 elements are in a strictly increasing order.
12 Return the index of value, or -1 if the value
13 doesn't exist in the list."""
14
15 def binary_search(input_array, value):
16     test_array = input_array
17     current_index = len(input_array)//2
18     input_index = current_index
19
20     found_value = test_array[current_index]
21     while(len(test_array)>1 and found_value!=value):
22         if(found_value<value):
23             test_array = test_array[current_index:]
24             current_index = len(test_array)//2
25             input_index += current_index
26             found_value = input_array[input_index]
27         else:
28             test_array = test_array[0:current_index]
29             current_index = len(test_array)//2
30             # divmod needed to be used instead of round() since the behavior
31             # for .5 changed from rounding up to rounding down in Python 3
32             q, r = divmod(len(test_array), 2.0)
33             input_index = int(input_index - q - r)
34             found_value = input_array[input_index]
35     else:
36         if(found_value==value):
37             return input_index
38
39     return -1
40
41 def linear_search(a, x):
42     for i in range(len(a)):
43         if a[i] == x:
44             return i
45     return -1
46
47 # compare naive algorithm linear search vs. binary search results
48 def stress_test(n, m):
49     test_cond = True
50     while(test_cond):
51         a = []
52         for i in range(n):
53             a.append(random.randint(0, 10**9))
54         a.sort()
55         for i in range(m):
56             b = random.randint(0, n-1)
57             print([linear_search(a, a[b]), binary_search(a, a[b])])
58             # stops if the searches do not give identical answers
59             if(linear_search(a, a[b]) != binary_search(a, a[b])):
60                 test_cond = False
61                 print('broke here!')
62                 break
63
64
65
66 stress_test(100, 100000)
67
68
69
70 #test_list = [1,3,9,11,15,19,29, 35, 36, 37]
71 #test_val1 = 25
```

```
72 #test_val2 = 15
73 #print(binary_search(test_list, test_val1))
74 #print(binary_search(test_list, test_val2))
75 #print(binary_search(test_list, 11))
76
```

```
1 # given array a and need to find value x
2 # left and right correspond to initial indices of array a bounding the search
3 # segment of array a above and below, respectively
4 def binary_search_recursive(a, x, left=0, right=(len(a)-1)):
5     """Recursive Binary Search algorithm implemented using list indexing"""
6     index = (left+right)//2
7     if a[index]==x:
8         return index
9     elif x>(a[right]) or x<(a[left]): # first case where x is not in the list!
10        return -1
11    elif left==right: # case where search is complete and no value x not found
12        return -1
13    elif left==right-1: # case where there are only two numbers left, check both!
14        left = right
15        return binary_search_recursive(a, x, left, right)
16    elif a[index]<x:
17        left = index
18        return binary_search_recursive(a, x, left, right)
19    elif a[index]>x:
20        right = index
21        return binary_search_recursive(a, x, left, right)
22
```

Binary Search Recursive:

```
1 def binarySearch(arr, searchValue):
2     low = 0
3     high = len(arr) - 1
4     while low <= high:
5         mid = (low + high) // 2
6         if arr[mid] < searchValue:
7             low = mid + 1
8         elif arr[mid] > searchValue:
9             high = mid - 1
10        else:
11            return True
12
13    return False
14
15
16 def binarySearchRec(arr, search_value):
17     if len(arr) == 0:
18         return False
19
```

```

1 """
2 Given an array where elements are sorted in ascending order,
3 convert it to a height balanced BST.
4 """
5
6
7 class TreeNode(object):
8     def __init__(self, x):
9         self.val = x
10        self.left = None
11        self.right = None
12
13
14 def array_to_bst(nums):
15     if not nums:
16         return None
17     mid = len(nums) // 2
18     node = TreeNode(nums[mid])
19     node.left = array_to_bst(nums[:mid])
20     node.right = array_to_bst(nums[mid + 1 :])
21
22     return node
23

```

```

1 """
2 Implement Binary Search Tree. It has method:
3     1. Insert
4     2. Search
5     3. Size
6     4. Traversal (Preorder, Inorder, Postorder)
7 """
8
9 import unittest
10
11
12 class Node(object):
13     def __init__(self, data):
14         self.data = data
15         self.left = None
16         self.right = None
17
18
19 class BST(object):
20     def __init__(self):
21         self.root = None
22
23     def get_root(self):
24         return self.root
25
26 """
27     Get the number of elements
28     Using recursion. Complexity O(logN)
29 """
30
31     def size(self):
32         return self.recur_size(self.root)
33
34     def recur_size(self, root):
35         if root is None:
36             return 0
37         else:
38             return 1 + self.recur_size(root.left) + self.recur_size(root.right)
39
40 """
41     Search data in bst
42     Using recursion. Complexity O(logN)
43 """
44
45     def search(self, data):
46         return self.recur_search(self.root, data)
47
48     def recur_search(self, root, data):
49         if root is None:

```

```

50         return False
51     if root.data == data:
52         return True
53     elif data > root.data: # Go to right root
54         return self.recur_search(root.right, data)
55     else: # Go to left root
56         return self.recur_search(root.left, data)
57
58 """
59     Insert data in bst
60     Using recursion. Complexity O(logN)
61 """
62
63 def insert(self, data):
64     if self.root:
65         return self.recur_insert(self.root, data)
66     else:
67         self.root = Node(data)
68         return True
69
70 def recur_insert(self, root, data):
71     if root.data == data: # The data is already there
72         return False
73     elif data < root.data: # Go to left root
74         if root.left: # If left root is a node
75             return self.recur_insert(root.left, data)
76         else: # left root is a None
77             root.left = Node(data)
78             return True
79     else: # Go to right root
80         if root.right: # If right root is a node
81             return self.recur_insert(root.right, data)
82         else:
83             root.right = Node(data)
84             return True
85
86 """
87     Preorder, Postorder, Inorder traversal bst
88 """
89
90 def preorder(self, root):
91     if root:
92         print(str(root.data), end=" ")
93         self.preorder(root.left)
94         self.preorder(root.right)
95
96 def inorder(self, root):
97     if root:
98         self.inorder(root.left)
99         print(str(root.data), end=" ")
100        self.inorder(root.right)
101
102 def postorder(self, root):
103     if root:
104         self.postorder(root.left)
105         self.postorder(root.right)
106         print(str(root.data), end=" ")
107
108
109 """
110     The tree is created for testing:
111
112           10
113           /   \
114          6    15
115          / \   / \
116         4   9  12  24
117         /       / \
118        7       20  30
119                 /
120                 18
121 """
122
123
124 class TestSuite(unittest.TestCase):
125     def setUp(self):

```

```

126     self.tree = BST()
127     self.tree.insert(10)
128     self.tree.insert(15)
129     self.tree.insert(6)
130     self.tree.insert(4)
131     self.tree.insert(9)
132     self.tree.insert(12)
133     self.tree.insert(24)
134     self.tree.insert(7)
135     self.tree.insert(20)
136     self.tree.insert(30)
137     self.tree.insert(18)
138
139 def test_search(self):
140     self.assertTrue(self.tree.search(24))
141     self.assertFalse(self.tree.search(50))
142
143 def test_size(self):
144     self.assertEqual(11, self.tree.size())
145
146
147 if __name__ == "__main__":
148     unittest.main()
149

```

Delete Node

```

1 """
2 Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node
3
4 Basically, the deletion can be divided into two stages:
5
6 Search for a node to remove.
7 If the node is found, delete the node.
8 Note: Time complexity should be O(height of tree).
9
10 Example:
11
12 root = [5,3,6,2,4,null,7]
13 key = 3
14
15      5
16      / \
17      3   6
18      / \   \
19     2   4   7
20
21 Given key to delete is 3. So we find the node with value 3 and delete it.
22
23 One valid answer is [5,4,6,2,null,null,7], shown in the following BST.
24
25      5
26      / \
27      4   6
28      /       \
29     2         7
30
31 Another valid answer is [5,2,6,null,4,null,7].
32
33      5
34      / \
35      2   6
36      \   \
37      4   7
38 """
39
40
41 class Solution(object):
42     def delete_node(self, root, key):

```

```

43     """
44     :type root: TreeNode
45     :type key: int
46     :rtype: TreeNode
47     """
48     if not root:
49         return None
50
51     if root.val == key:
52         if root.left:
53             # Find the right most leaf of the left sub-tree
54             left_right_most = root.left
55             while left_right_most.right:
56                 left_right_most = left_right_most.right
57             # Attach right child to the right of that leaf
58             left_right_most.right = root.right
59             # Return left child instead of root, a.k.a delete root
60             return root.left
61         else:
62             return root.right
63     # If left or right child got deleted, the returned root is the child of the deleted node.
64     elif root.val > key:
65         root.left = self.deleteNode(root.left, key)
66     else:
67         root.right = self.deleteNode(root.right, key)
68
69     return root

```

Another:

```

1 def binary_search(arr, x):
2     start= 0
3     end = len(arr) - 1
4     mid = 0
5     while start<= end:
6         mid = (start+ end) // 2
7         # If x is greater, search in right array
8         if arr[mid] < x:
9             start = mid + 1
10
11        # If x is smaller, search in left array
12        elif arr[mid] > x:
13            end= mid - 1
14        # if x is present at mid
15        else:
16            return mid
17
18
19    # when we reach at the end of array, then the element was not present
20    return -1
21
22
23 arr = [ ]
24 n=int(input("Enter size of array : "))
25 print("Enter array elements : ")
26 for i in range(n):
27     e=int(input())
28     arr.append(e)
29 x = int(input("Enter element to search "))
30 ans = binary_search(arr, x)
31 if(ans== -1):
32     print("Element not found")
33 else:
34     print("Element found at ",ans)

```

Binary Tree

```
1 # Implement a Binary Search Tree (BST) that can insert values and check if
2 # values are present
3
4 class Node(object):
5     def __init__(self, value):
6         self.value = value
7         self.left = None
8         self.right = None
9
10 class BST(object):
11     def __init__(self, root):
12         self.root = Node(root)
13
14     def insert(self, new_val):
15         if(self.root.left==None):
16             if(self.root.value>new_val):
17                 self.root.left = Node(new_val)
18             elif(self.root.right==None):
19                 if(self.root.value<new_val):
20                     self.root.right = Node(new_val)
21             else:
22                 current = self.root
23                 while(current.left!=None or current.right!=None):
24                     if(current.value>new_val):
25                         current = current.left
26                     else:
27                         current = current.right
28
29                 if(current.left==None):
30                     current.left = Node(new_val)
31                 else:
32                     current.right = Node(new_val)
33
34     def search(self, find_val):
35         if(self.root.left==None and self.root.right==None and self.root.value!=find_val):
36             return False
37         else:
38             current = self.root
39             val_possible = True
40             while(val_possible):
41                 if(current.value==find_val):
42                     return True
43                 if(current.value<find_val):
44                     current = current.right
45                 else:
46                     current = current.left
47                 if(current==None):
48                     return False
49                 if(current.value<find_val and (current.right==None or current.right>find_val)):
50                     return False
51                 if(current.value>find_val and (current.left==None or current.left<find_val)):
52                     return False
53
54 # Set up tree
55 tree = BST(4)
56
57 # Insert elements
58 tree.insert(2)
59 tree.insert(1)
60 tree.insert(3)
61 tree.insert(5)
62
63 # Check search
64 # Should be True
65 print tree.search(4)
66 # Should be False
67 print tree.search(6)
68
```

```

1  class Solution(object):
2      def topKFrequent(self, nums, k):
3          number_frequency = {}
4          frequency_list = []
5          for i in nums:
6              if i not in number_frequency:
7                  number_frequency[i] = 1
8              else:
9                  number_frequency[i] += 1
10         for key, value in number_frequency.items():
11             if value not in frequency_list:
12                 frequency_list[value] = [key]
13             else:
14                 frequency_list[value].append(key)
15         result = []
16         for i in range(len(nums), 0, -1):
17             if i in frequency_list:
18                 result.extend(frequency_list[i])
19             if len(result) >= k:
20                 break
21         return result
22
23
24 ob1 = Solution()
25 print(ob1.topKFrequent([1, 1, 1, 1, 2, 2, 3, 3, 3], 2))
26
27
28
29

```

Balanced Binary Tree

Balanced Binary Tree

Given a binary tree class that looks like this:

```

1  class BinaryTreeNode {
2      constructor(value) {
3          this.value = value;
4          this.left = null;
5          this.right = null;
6      }
7
8      insertLeft(value) {
9          this.left = new BinaryTreeNode(value);
10         return this.left;
11     }
12
13     insertRight(value) {
14         this.right = new BinaryTreeNode(value);
15         return this.right;
16     }
17 }

```

write a function that checks to see if a given binary tree is perfectly balanced, meaning all leaf nodes are located at the same depth. Your function should return `true` if the tree is perfectly balanced and `false` otherwise.

Analyze the time and space complexity of your function.

JS Solution:

```
1  /*
2   * A recursive solution
3   * How would you solve this iteratively?
4   */
5  const checkBalanced = (rootNode) => {
6      // An empty tree is balanced by default
7      if (!rootNode) return true;
8      // recursive helper function to check the min depth of the tree
9      const minDepth = (node) => {
10          if (!node) return 0;
11          return 1 + Math.min(minDepth(node.left), minDepth(node.right));
12      };
13      // recursive helper function to check the max depth of the tree
14      const maxDepth = (node) => {
15          if (!node) return 0;
16          return 1 + Math.max(maxDepth(node.left), maxDepth(node.right));
17      };
18
19      return maxDepth(rootNode) - minDepth(rootNode) === 0;
20  };
21
22 /* Some console.log tests */
23 class BinaryTreeNode {
24     constructor(value) {
25         this.value = value;
26         this.left = null;
27         this.right = null;
28     }
29
30     insertLeft(value) {
31         this.left = new BinaryTreeNode(value);
32         return this.left;
33     }
34
35     insertRight(value) {
36         this.right = new BinaryTreeNode(value);
37         return this.right;
38     }
39 }
40
41 const root = new BinaryTreeNode(5);
42 console.log(checkBalanced(root)); // should print true
43
44 root.insertLeft(10);
45 console.log(checkBalanced(root)); // should print false
46
47 root.insertRight(11);
48 console.log(checkBalanced(root)); // should print true;
49
```

```
1 # A recursive solution
2 # How would you solve this iteratively?
3
4
5 def checkBalanced(rootNode):
6     # An empty tree is balanced by default
7     if rootNode == None:
8         return True
9
10    # recursive helper function to check the min depth of the tree
11    def minDepth(node):
12        if node == None:
13            return 0
14        return 1 + min(minDepth(node.left), minDepth(node.right))
15
16    # recursive helper function to check the max depth of the tree
```

```

17     def maxDepth(node):
18         if node == None:
19             return 0
20         return 1 + max(maxDepth(node.left), maxDepth(node.right))
21
22     return maxDepth(rootNode) - minDepth(rootNode) == 0
23
24
25 # Some console.log tests
26 class BinaryTreeNode:
27     def __init__(self, value):
28         self.value = value
29         self.left = None
30         self.right = None
31
32     def insertLeft(self, value):
33         self.left = BinaryTreeNode(value)
34         return self.left
35
36     def insertRight(self, value):
37         self.right = BinaryTreeNode(value)
38         return self.right
39
40
41 root = BinaryTreeNode(5)
42 print(checkBalanced(root)) # should print True
43
44 root.insertLeft(10)
45 print(checkBalanced(root)) # should print False
46
47 root.insertRight(11)
48 print(checkBalanced(root)) # should print True
49

```

Binary Search Tree from Sorted Array

Given an array that is sorted in ascending order containing unique integer elements, write a function that receives the sorted array as input and creates a valid binary search tree with minimal height.

For example, given an array `[1, 2, 3, 4, 5, 6, 7]`, your function should return a binary search tree with the form

```

1           4
2           /   \
3           2     6
4          / \   / \
5         1   3  5   7

```

Note that when we say "binary search tree" in this case, we're just talking about a tree that exhibits the expected *form* of a binary search tree. The tree in this case won't have an `insert` method that does the work of receiving a value and then inserting it in a valid spot in the binary search tree. Your function should place the values in valid spots that adhere to the rules of binary search trees, while also seeking to minimize the overall height of the tree.

Here's a `BinaryTreeNode` class that you can use to construct a binary search tree:

```

1 class BinaryTreeNode:
2     def __init__(self, value):
3         self.value = value
4         self.left = None
5         self.right = None

```

Analyze the time and space complexity of your solution.

Create a Minimal Height BST from Sorted Array

Understanding the Problem

This problem asks us to create a valid binary search tree from a sorted array of integers. More specifically, the resulting binary search tree needs to be of *minimal height*. Our function should return the root node of the created binary search tree.

From the given example where the input is `[1, 2, 3, 4, 5, 6, 7]`, the expected answer is a binary search tree of height 3. This is the minimal height that can be achieved for an array of 7 seven elements. Try as we might, there's no way to construct a binary search tree containing all of these elements that has a shorter height.

Coming Up with a First Pass

A straightforward way to do this would be to take the first element of our array, call that the root, and then iterate through the rest of our array, adding those elements as nodes in the binary search tree. In pseudocode, that might look something like this:

```
1 def create_min_height_bst(sorted_arr):
2     root = BinaryTreeNode(sorted_arr[0])
3
4     for elem in sorted_arr:
5         root.insert(elem)
6
7     return root
8
```

```
1 function createMinHeightBST(sortedArray) {
2     const left = 0;
3     const right = sortedArray.length - 1;
4
5     return recHelper(sortedArray, left, right);
6 }
7
8 function recHelper(sortedArray, left, right) {
9     if (left > right) {
10         return null;
11     }
12
13     const midpoint = math.floor(right - left) / 2 + left;
14     const root = new BinaryTreeNode(sortedArray[midpoint]);
15
16     root.left = recHelper(sortedArray, left, midpoint - 1);
17     root.right = recHelper(sortedArray, midpoint + 1, right);
18
19     return root;
20 }
21
22 class BinaryTreeNode {
23     constructor(value) {
24         this.value = value;
25         this.left = null;
26         this.right = null;
27     }
28 }
29
```

```

30 function isBST(root, minBound, maxBound) {
31   if (root === null) {
32     return true;
33   }
34
35   if (root.value < minBound || root.value > maxBound) {
36     return false;
37   }
38
39   const left = isBST(root.left, minBound, root.value - 1);
40   const right = isBST(root.right, root.value + 1, maxBound);
41
42   return left && right;
43 }
44
45 function findBSTMaxHeight(node) {
46   if (node === null) {
47     return 0;
48   }
49
50   return (
51     1 + Math.max(findBSTMaxHeight(node.left), findBSTMaxHeight(node.right))
52 );
53 }
54
55 function isBSTMinHeight(root, N) {
56   const height = findBSTMaxHeight(root);
57   const shouldEqual = Math.floor(Math.log2(N)) + 1;
58
59   return height === shouldEqual;
60 }
61
62 function countBSTNodes(root, count) {
63   if (root === null) {
64     return count;
65   }
66
67   countBSTNodes(root.left, count);
68   count++;
69   countBSTNodes(root.right, count);
70 }
71
72 // Some tests
73 let sortedArray = [1, 2, 3, 4, 5, 6, 7];
74 let bst = createMinHeightBST(sortedArray);
75
76 console.log(isBST(bst, -Infinity, Infinity));
77 console.log(isBSTMinHeight(bst, sortedArray.length));
78
79 sortedArray = [4, 10, 11, 18, 42, 43, 47, 49, 55, 67, 79, 89, 90, 95, 98, 100];
80 bst = createMinHeightBST(sortedArray);
81
82 console.log(isBST(bst, -Infinity, Infinity));
83 console.log(isBSTMinHeight(bst, sortedArray.length));
84

```

```

1 import math
2
3
4 def create_min_height_bst(sorted_array):
5     left = 0
6     right = len(sorted_array) - 1
7
8     return rec_helper(sorted_array, left, right)
9
10
11 def rec_helper(sorted_array, left, right):
12     if left > right:
13         return None
14
15     midpoint = ((right - left) // 2) + left
16     root = BinaryTreeNode(sorted_array[midpoint])
17

```

```

18     root.left = rec_helper(sorted_array, left, midpoint - 1)
19     root.right = rec_helper(sorted_array, midpoint + 1, right)
20
21     return root
22
23
24 class BinaryTreeNode:
25     def __init__(self, value):
26         self.value = value
27         self.left = None
28         self.right = None
29
30
31 # Helper function to validate that the created tree is a valid BST
32 def is_BST(root, min_bound, max_bound):
33     if root is None:
34         return True
35
36     if root.value < min_bound or root.value > max_bound:
37         return False
38
39     left = is_BST(root.left, min_bound, root.value - 1)
40     right = is_BST(root.right, root.value + 1, max_bound)
41
42     return left and right
43
44
45 # Helper function to check the max height of a BST
46 def find_bst_max_height(node):
47     if node is None:
48         return 0
49
50     return 1 + max(find_bst_max_height(node.left), find_bst_max_height(node.right))
51
52
53 # Helper function to validate that the given BST exhibits the min height
54 def is_bst_min_height(root, N):
55     bst_max_height = find_bst_max_height(root)
56     should_equal = math.floor(math.log2(N)) + 1
57
58     return bst_max_height == should_equal
59
60
61 # Helper function to count the number of nodes for a given BST
62 def count_bst_nodes(root, count):
63     if root is None:
64         return count
65
66     count_bst_nodes(root.left, count)
67     count += 1
68     count_bst_nodes(root.right, count)
69
70
71 # Some tests
72 sorted_array = [1, 2, 3, 4, 5, 6, 7]
73 bst = create_min_height_bst(sorted_array)
74
75 print(is_BST(bst, float("-inf"), float("inf"))) # should print true
76 print(is_bst_min_height(bst, len(sorted_array))) # should print true
77
78 sorted_array = [4, 10, 11, 18, 42, 43, 47, 49, 55, 67, 79, 89, 90, 95, 98, 100]
79 bst = create_min_height_bst(sorted_array)
80
81 print(is_BST(bst, float("-inf"), float("inf"))) # should print true
82 print(is_bst_min_height(bst, len(sorted_array))) # should print true
83

```

Another BST Implementation:

```
1 class BinarySearchTree:
2     def __init__(self, value):
3         self.value = value
4         self.left = None
5         self.right = None
6
7     def insert(self, value):
8         pass
9
10    def contains(self, target):
11        pass
12
13    def get_max(self):
14        pass
15
16    def for_each(self, cb):
17        pass
18
```

```
1 import unittest
2 import random
3 from binary_search_tree import BinarySearchTree
4
5 class BinarySearchTreeTests(unittest.TestCase):
6     def setUp(self):
7         self.bst = BinarySearchTree(5)
8
9     def test_insert(self):
10         self.bst.insert(2)
11         self.bst.insert(3)
12         self.bst.insert(7)
13         self.bst.insert(6)
14         self.assertEqual(self.bst.left.right.value, 3)
15         self.assertEqual(self.bst.right.left.value, 6)
16
17     def test_contains(self):
18         self.bst.insert(2)
19         self.bst.insert(3)
20         self.bst.insert(7)
21         self.assertTrue(self.bst.contains(7))
22         self.assertFalse(self.bst.contains(8))
23
24     def test_get_max(self):
25         self.assertEqual(self.bst.get_max(), 5)
26         self.bst.insert(30)
27         self.assertEqual(self.bst.get_max(), 30)
28         self.bst.insert(300)
29         self.bst.insert(3)
30         self.assertEqual(self.bst.get_max(), 300)
31
32     def test_for_each(self):
33         arr = []
34         cb = lambda x: arr.append(x)
35
36         v1 = random.randint(1, 101)
37         v2 = random.randint(1, 101)
38         v3 = random.randint(1, 101)
39         v4 = random.randint(1, 101)
40         v5 = random.randint(1, 101)
41
42         self.bst.insert(v1)
43         self.bst.insert(v2)
44         self.bst.insert(v3)
45         self.bst.insert(v4)
46         self.bst.insert(v5)
47
48         self.bst.for_each(cb)
49
50         self.assertTrue(5 in arr)
51         self.assertTrue(v1 in arr)
52         self.assertTrue(v2 in arr)
53         self.assertTrue(v3 in arr)
```

```
54     self.assertTrue(v4 in arr)
55     self.assertTrue(v5 in arr)
56
57
58 if __name__ == '__main__':
59     unittest.main()
60
```

Binary Search Tree

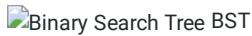
Implementation

```
1 # Implement a Binary Search Tree (BST) that can insert values and check if
2 # values are present
3
4 class Node(object):
5     def __init__(self, value):
6         self.value = value
7         self.left = None
8         self.right = None
9
10 class BST(object):
11     def __init__(self, root):
12         self.root = Node(root)
13
14     def insert(self, new_val):
15         if(self.root.left==None):
16             if(self.root.value>new_val):
17                 self.root.left = Node(new_val)
18         elif(self.root.right==None):
19             if(self.root.value<new_val):
20                 self.root.right = Node(new_val)
21         else:
22             current = self.root
23             while(current.left!=None or current.right!=None):
24                 if(current.value>new_val):
25                     current = current.left
26                 else:
27                     current = current.right
28
29                 if(current.left==None):
30                     current.left = Node(new_val)
31                 else:
32                     current.right = Node(new_val)
33
34     def search(self, find_val):
35         if(self.root.left==None and self.root.right==None and self.root.value!=find_val):
36             return False
37         else:
38             current = self.root
39             val_possible = True
40             while(val_possible):
41                 if(current.value==find_val):
42                     return True
43                 if(current.value<find_val):
44                     current = current.right
45                 else:
46                     current = current.left
47                 if(current==None):
48                     return False
49                 if(current.value<find_val and (current.right==None or current.right>find_val)):
50                     return False
51                 if(current.value>find_val and (current.left==None or current.left<find_val)):
52                     return False
53
54     # Set up tree
55     tree = BST(4)
56
57     # Insert elements
58     tree.insert(2)
59     tree.insert(1)
60     tree.insert(3)
61     tree.insert(5)
62
63     # Check search
64     # Should be True
65     print tree.search(4)
66     # Should be False
67     print tree.search(6)
68
```

Preorder Operations

```
1  class Node(object):
2      def __init__(self, value):
3          self.value = value
4          self.left = None
5          self.right = None
6
7  class BinaryTree(object):
8      def __init__(self, root):
9          self.root = Node(root)
10
11     def preorder_search(self, start, find_val):
12         """Helper method - use this to create a
13         recursive search solution."""
14         if(start.value == find_val):
15             return True
16         if(start.left!=None):
17             left_result = self.preorder_search(start.left, find_val)
18         else:
19             left_result = False
20         if(start.right!=None and left_result!=True):
21             right_result = self.preorder_search(start.right, find_val)
22         else:
23             right_result = False
24
25         if(left_result==True or right_result==True):
26             return True
27         else:
28             return False
29
30     def preorder_print(self, start, traversal):
31         """Helper method - use this to create a
32         recursive print solution."""
33         traversal += '-' + str(start.value)
34         left_nums = ''
35         right_nums = ''
36         if(start.left!=None):
37             traversal = self.preorder_print(start.left, traversal)
38         if(start.right!=None):
39             traversal = self.preorder_print(start.right, traversal)
40         return traversal
41
42     def search(self, find_val):
43         """Return True if the value
44         is in the tree, return
45         False otherwise."""
46         return self.preorder_search(self.root, find_val)
47         #print(self.preorder_search(self.root, find_val))
48
49     def print_tree(self):
50         """Print out all tree nodes
51         as they are visited in
52         a pre-order traversal."""
53         all_nodes = self.preorder_print(self.root, '')
54         all_nodes = all_nodes[1:]
55         return all_nodes
56
57
58 # Set up tree
59 tree = BinaryTree(1)
60 tree.root.left = Node(2)
61 tree.root.right = Node(3)
62 tree.root.left.left = Node(4)
63 tree.root.left.right = Node(5)
64
65 # Test search
66 # Should be True
67 print(tree.search(4))
68 # Should be False
69 print(tree.search(6))
70
71 # Test print_tree
72 # Should be 1-2-4-5-3
```

```
73 print(tree.print_tree())
74
```



Pros of a BST

- When balanced, a BST provides lightning-fast $O(\log(n))$ insertions, deletions, and lookups.
- Binary search trees are pretty simple. An ordinary BST, unlike a balanced tree like a red-black tree, requires very little code to get running.

Cons of a BST

- Slow for a brute-force search. If you need to iterate over each node, you might have more success with an array.
- When the tree becomes unbalanced, all fast $O(\log(n))$ operations quickly degrade to $O(n)$.
- Since pointers to whole objects are typically involved, a BST can require quite a bit more memory than an array, although this depends on the implementation.

Implementing a BST in Python

Step 1 – BSTNode Class

Our implementation won't use a `Tree` class, but instead just a `Node` class. Binary trees are really just a pointer to a root node that in turn connects to each child node, so we'll run with that idea.

First, we create a constructor:

```
1 class BSTNode:
2     def __init__(self, val=None):
3         self.left = None
4         self.right = None
5         self.val = val
```

We'll allow a value (`key`) to be provided, but if one isn't provided we'll just set it to `None`. We'll also initialize both children of the new node to `None`.

Step 2 – Insert

We need a way to insert new data. The `insert` method is as follows:

```
1 def insert(self, val):
2     if not self.val:
3         self.val = val
4         return
5
6     if self.val == val:
7         return
8
```

```

9     if val < self.val:
10        if self.left:
11            self.left.insert(val)
12        return
13        self.left = BSTNode(val)
14        return
15
16    if self.right:
17        self.right.insert(val)
18        return
19    self.right = BSTNode(val)Code language: Python (python)

```

If the node doesn't yet have a value, we can just set the given value and return. If we ever try to insert a value that also exists, we can also simply return as this can be considered a `noop`. If the given value is less than our node's value and we already have a left child then we recursively call `insert` on our left child. If we don't have a left child yet then we just make the given value our new left child. We can do the same (but inverted) for our right side.

Step 3 – Get Min and Get Max

```

1 def get_min(self):
2     current = self
3     while current.left is not None:
4         current = current.left
5     return current.val
6
7 def get_max(self):
8     current = self
9     while current.right is not None:
10        current = current.right
11    return current.valCode language: Python (python)

```

`getMin` and `getMax` are useful helper functions, and they're easy to write! They are simple recursive functions that traverse the edges of the tree to find the smallest or largest values stored therein.

Step 4 – Delete

```

1 def delete(self, val):
2     if self == None:
3         return self
4     if val < self.val:
5         self.left = self.left.delete(val)
6         return self
7     if val > self.val:
8         self.right = self.right.delete(val)
9         return self
10    if self.right == None:
11        return self.left
12    if self.left == None:
13        return self.right
14    min_larger_node = self.right
15    while min_larger_node.left:
16        min_larger_node = min_larger_node.left
17    self.val = min_larger_node.val
18    self.right = self.right.delete(min_larger_node.val)
19    return selfdef delete(self, val):
20    if self == None:
21        return self
22    if val < self.val:
23        if self.left:
24            self.left = self.left.delete(val)
25        return self
26    if val > self.val:
27        if self.right:
28            self.right = self.right.delete(val)
29        return self

```

```

30     if self.right == None:
31         return self.left
32     if self.left == None:
33         return self.right
34     min_larger_node = self.right
35     while min_larger_node.left:
36         min_larger_node = min_larger_node.left
37     self.val = min_larger_node.val
38     self.right = self.right.delete(min_larger_node.val)
39     return selfdef language: Python (python)
40
41     if self == None:
42         return self
43     if val < self.val:
44         self.left = self.left.delete(val)
45         return self
46     if val > self.val:
47         self.right = self.right.delete(val)
48         return self
49     if self.right == None:
50         return self.left
51     if self.left == None:
52         return self.right
53     min_larger_node = self.right
54     while min_larger_node.left:
55         min_larger_node = min_larger_node.left
56     self.val = min_larger_node.val
57     self.right = self.right.delete(min_larger_node.val)
      return selfCode language: Python (python)

```

The delete operation is one of the more complex ones. It is a recursive function as well, but it also returns the new state of the given node after performing the delete operation. This allows a parent whose child has been deleted to properly set its `left` or `right` data member to `None`.

Step 5 – Exists

The exists function is another simple recursive function that returns `True` or `False` depending on whether a given value already exists in the tree.

```

1 def exists(self, val):
2     if val == self.val:
3         return True
4
5     if val < self.val:
6         if self.left == None:
7             return False
8         return self.left.exists(val)
9
10    if self.right == None:
11        return False
12    return self.right.exists(val)Code language: Python (python)

```

Step 6 – Inorder

It's useful to be able to print out the tree in a readable format. The `inorder` method prints the values in the tree in the order of their keys.

```

1 def inorder(self, vals):
2     if self.left is not None:
3         self.left.inorder(vals)
4     if self.val is not None:
5         vals.append(self.val)
6     if self.right is not None:
7         self.right.inorder(vals)
8     return valsCode language: Python (python)

```

Step 7 – Preorder

```
1 def preorder(self, vals):
2     if self.val is not None:
3         vals.append(self.val)
4     if self.left is not None:
5         self.left.preorder(vals)
6     if self.right is not None:
7         self.right.preorder(vals)
8     return valsCode language: Python (python)
```

Step 8 – Postorder

```
1 def postorder(self, vals):
2     if self.left is not None:
3         self.left.postorder(vals)
4     if self.right is not None:
5         self.right.postorder(vals)
6     if self.val is not None:
7         vals.append(self.val)
8     return valsCode language: Python (python)
```

Using the BST

```
1 def main():
2     nums = [12, 6, 18, 19, 21, 11, 3, 5, 4, 24, 18]
3     bst = BSTNode()
4     for num in nums:
5         bst.insert(num)
6     print("preorder:")
7     print(bst.preorder([]))
8     print("#")
9
10    print("postorder:")
11    print(bst.postorder([]))
12    print("#")
13
14    print("inorder:")
15    print(bst.inorder([]))
16    print("#")
17
18    nums = [2, 6, 20]
19    print("deleting " + str(nums))
20    for num in nums:
21        bst.delete(num)
22    print("#")
23
24    print("4 exists:")
25    print(bst.exists(4))
26    print("12 exists:")
27    print(bst.exists(2))
28    print("18 exists:")
29    print(bst.exists(12))
30    print("18 exists:")
31    print(bst.exists(18))Code language: Python (python)
```

Full Binary Search Tree in Python

```
1 class BSTNode:
2     def __init__(self, val=None):
3         self.left = None
```

```

4         self.right = None
5         self.val = val
6
7     def insert(self, val):
8         if not self.val:
9             self.val = val
10            return
11
12        if self.val == val:
13            return
14
15        if val < self.val:
16            if self.left:
17                self.left.insert(val)
18            return
19            self.left = BSTNode(val)
20            return
21
22        if self.right:
23            self.right.insert(val)
24            return
25            self.right = BSTNode(val)
26
27    def get_min(self):
28        current = self
29        while current.left is not None:
30            current = current.left
31        return current.val
32
33    def get_max(self):
34        current = self
35        while current.right is not None:
36            current = current.right
37        return current.val
38
39    def delete(self, val):
40        if self == None:
41            return self
42        if val < self.val:
43            if self.left:
44                self.left = self.left.delete(val)
45            return self
46        if val > self.val:
47            if self.right:
48                self.right = self.right.delete(val)
49            return self
50        if self.right == None:
51            return self.left
52        if self.left == None:
53            return self.right
54        min_larger_node = self.right
55        while min_larger_node.left:
56            min_larger_node = min_larger_node.left
57        self.val = min_larger_node.val
58        self.right = self.right.delete(min_larger_node.val)
59        return self
60
61    def exists(self, val):
62        if val == self.val:
63            return True
64
65        if val < self.val:
66            if self.left == None:
67                return False
68            return self.left.exists(val)
69
70        if self.right == None:
71            return False
72        return self.right.exists(val)
73
74    def preorder(self, vals):
75        if self.val is not None:
76            vals.append(self.val)
77        if self.left is not None:
78            self.left.preorder(vals)
79        if self.right is not None:

```

```

80         self.right.preorder(vals)
81     return vals
82
83     def inorder(self, vals):
84         if self.left is not None:
85             self.left.inorder(vals)
86         if self.val is not None:
87             vals.append(self.val)
88         if self.right is not None:
89             self.right.inorder(vals)
90     return vals
91
92     def postorder(self, vals):
93         if self.left is not None:
94             self.left.postorder(vals)
95         if self.right is not None:
96             self.right.postorder(vals)
97         if self.val is not None:
98             vals.append(self.val)
99     return vals
```

Code language: Python (python)

Where would you use a binary search tree in real life?

There are many applications of binary search trees in real life, and one of the most common use-cases is in storing indexes and keys in a database. For example, in MySQL or PostgresQL when you create a primary key column, what you're really doing is creating a binary tree where the keys are the values of the column, and those nodes point to database rows. This lets the application easily search database rows by providing a key. For example, getting a user record by the `email` primary key.

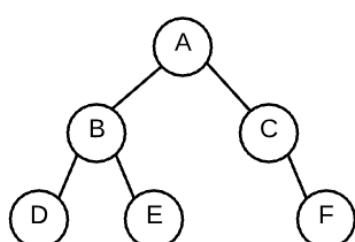
There are many applications of binary search trees in real life, and one of the most common use cases is storing indexes and keys in a database. For example, when you create a primary key column in MySQL or PostgresQL, you create a binary tree where the keys are the values of the column and the nodes point to database rows. This allows the application to easily search for database rows by specifying a key, for example, to find a user record using the email primary key.

Other common uses include:

- Pathfinding algorithms in videogames (A*) use BSTs
- File compression using a Huffman encoding scheme uses a binary search tree
- Rendering calculations – Doom (1993) was famously the first game to use a BST
- Compilers for low-level coding languages parse syntax using a BST
- Almost every database in existence uses BSTs for key lookups

Example Binary Tree

Visual Aid



Example Code

```
1 class TreeNode {
2     constructor(val) {
3         this.val = val;
4         this.left = null;
5         this.right = null;
6     }
7 }
8
9 let a = new TreeNode("a");
10 let b = new TreeNode("b");
11 let c = new TreeNode("c");
12 let d = new TreeNode("d");
13 let e = new TreeNode("e");
14 let f = new TreeNode("f");
15
16 a.left = b;
17 a.right = c;
18 b.left = d;
19 b.right = e;
20 c.right = f;
```

Terms

- tree - graph with no cycles
- binary tree - tree where nodes have at most 2 nodes
- root - the ultimate parent, the single node of a tree that can access every other node through edges; by definition the root will not have a parent
- internal node - a node that has children
- leaf - a node that does not have any children
- path - a series of nodes that can be traveled through edges - for example A, B, E is a path through the above tree

Search Patterns

- Breadth First Search - Check all nodes at a level before moving down a level
 - Think of this of searching horizontally in rows
- Depth First Search - Check the depth as far as it goes for one child, before moving on to the next child.
 - Think of this as searching vertically in diagonals
 - Pre-Order Traversal - Access the data of the current node, recursively visit the left sub tree, recursively visit the right sub tree
 - All the way to the left, top down, going right after other options have already been logged.
 - In-Order Traversal - Recursively visit the left sub tree, access the data of the current node, recursively visit the right sub tree
 - In the order they were the "current root", the actual return order of the recursive calls.
 - Post-Order Traversal - Recursively visit the left sub tree, recursively visit the right sub tree, access the data of the current node.
 - Starting with the bottom most nodes working up through the tree

Constraints

- Binary trees have at most two children per node
- Given any node of the tree, the values on the left must be strictly less than that node
- Given any node of the tree, the values on the right must be strictly greater than or equal to that node
- Given these constraints a binary tree is necessarily a sorted data structure

- The worst binary trees devolve into a linked list, the best are height balanced (think branching).
-

PseudoCode For Insertion

- Create a new node
- Start at the root
 - Check if there is a root
 - If not the root becomes the new node
 - If there is a root check if the value of the new node is equal to, greater than, or less than the value of the root
 - If it is greater or equal to
 - Check to see if there is a node to the right
 - If there is, move to the new node and continue with the node to the right as the subtree root
 - If there is not, add the new node as the right property of the current node
 - If it is smaller
 - Check to see if there is a node to the left
 - If there is, move to the new node and continue with the node to the left as the subtree root
 - If there is not, add the new node as the left property of the current node

PseudoCode For Search Of A single Item

- Start at the root
 - Check if there is a root
 - If not, there is nothing in the tree, so the search is over
 - If there is a root, check if the value of the root is equal to, greater than, or less than the value we're looking for;
 - If it is equal to the value
 - We found what we are searching for
 - If it is less than the value
 - Check to see if there is a node to the left
 - If there isn't
 - the value isn't in our tree
 - If there is
 - repeat these steps with the node to the left as the new subtree root
 - If it is greater than the value
 - Check to see if there is a node to the right
 - If there isn't
 - the value isn't in our tree
 - If there is
 - repeat these steps with the node to the right as the new subtree root

PseudoCode For Breadth First Search Traversal

- Create a queue class or use an array
 - Create a variable to store the values of the nodes visited
 - Place the root in the queue
 - Loop as many times as there are items in the queue
 - Dequeue a node
 - If there is a left value to the node dequeued, add it to the queue
 - If there is a right value to the node dequeued, add it to the queue
 - Push the node's value into the variable that stores nodes visited
-

PseudoCode For Depth First Search Traversal

Pre-Order

Iterative

- Create a stack class or use an array
- Push the root into the stack
- Create a variable to store the values of the nodes visited
- Do this as long as there is something on the stack
 - Pop a node from the stack
 - Push that nodes value into the variable that stores nodes visited.
 - If there is a node to the right push it into the stack
 - If there is a node to the left push it into the stack
- Return the variable storing the values

Recursive

- Create a variable to store the current root
- Push the value of current root to the variable storing the values
- If the current root has a left property call the function on that the left property
- If the current root has a right property call the function on that the right property
- Spread the current root, the left values, and the right values

In-Order

Iterative

- Create a stack class or use an array
- Create a variable to store the current root
- Create a variable to store the values of the nodes visited
- Create a loop
 - While the current root exists
 - push the current root to the call stack
 - current root is equal to the left of current root
 - if the stack is empty break out of the loop
 - set a variable to equal the popped value of the stack
 - push that variable into the variable that stores values
 - set the current root to the right of the current loop
- Return the variable storing the values

Recursive

- Create a variable to store the current root
- Push the value of current root to the variable storing the values
- If the current root has a left property call the function on that the left property
- If the current root has a right property call the function on that the right property
- Spread the the left values, current root ,and the right values

Post-Order

Iterative

- Haven't figured this one out yet.

Recursive

- Create a variable to store the current root
 - Push the value of current root to the variable storing the values
 - If the current root has a left property call the function on that the left property
 - If the current root has a right property call the function on that the right property
 - Spread the the left values, the right values, and current root
-

Example Binary Search Tree

```
1  class TreeNode {
2      constructor(val) {
3          this.val = val;
4          this.left = null;
5          this.right = null;
6      }
7  }
8
9 class BST {
10    constructor() {
11        this.root = null;
12    }
13
14 //Insert a new node
15
16 recursiveInsert(val, currentNode = this.root) {
17     if (!this.root) {
18         this.root = new TreeNode(val);
19         return this;
20     }
21     if (val < currentNode.val) {
22         if (!currentNode.left) {
23             currentNode.left = new TreeNode(val);
24         } else {
25             this.insert(val, currentNode.left);
26         }
27     } else {
28         if (!currentNode.right) {
29             currentNode.right = new TreeNode(val);
30         } else {
31             this.insert(val, currentNode.right);
32         }
33     }
34 }
35
36 iterativeInsert(val, currentNode = this.root) {
37     if (!this.root) {
38         this.root = new TreeNode(val);
39         return this;
40     }
41     if (val < currentNode.val) {
42         if (!currentNode.left) {
43             currentNode.left = new TreeNode();
44         } else {
45             while (true) {
46                 if (val < currentNode.val) {
47                     if (!currentNode.left) {
48                         currentNode.left = new TreeNode();
49                         return this;
50                     } else {
51                         currentNode = currentNode.left;
52                     }
53                 } else {
54                     if (!currentNode.right) {
55                         currentNode.right = new TreeNode();
56                         return this;
57                     } else {
58                         currentNode = currentNode.right;
59                     }
60                 }
61             }
62         }
63     }
64 }
```

```

59         }
60     }
61   }
62 }
63 }
64 }
65
66 //Search the tree
67
68 searchRecur(val, currentNode = this.root) {
69   if (!currentNode) return false;
70   if (val < currentNode.val) {
71     return this.searchRecur(val, currentNode.left);
72   } else if (val > currentNode.val) {
73     return this.searchRecur(val, currentNode.right);
74   } else {
75     return true;
76   }
77 }
78
79 searchIter(val) {
80   let currentNode = this.root;
81   while (currentNode) {
82     if (val < currentNode.val) {
83       currentNode = currentNode.left;
84     } else if (val > currentNode.val) {
85       currentNode = currentNode.right;
86     } else {
87       return true;
88     }
89   }
90   return false;
91 }
92
93 // Maybe works, who knows, pulled it off the internet....
94
95 deleteNodeHelper(root, key) {
96   if (root === null) {
97     return false;
98   }
99   if (key < root.val) {
100     root.left = deleteNodeHelper(root.left, key);
101     return root;
102   } else if (key > root.val) {
103     root.right = deleteNodeHelper(root.right, key);
104     return root;
105   } else {
106     if (root.left === null && root.right === null) {
107       root = null;
108       return root;
109     }
110     if (root.left === null) return root.right;
111     if (root.right === null) return root.left;
112
113     let currNode = root.right;
114     while (currNode.left !== null) {
115       currNode = currNode.left;
116     }
117     root.val = currNode.val;
118     root.right = deleteNodeHelper(root.right, currNode.val);
119     return root;
120   }
121 }
122
123 //Recursive Depth First Search
124
125 preOrderTraversal(root) {
126   if (!root) return [];
127   let left = this.preOrderTraversal(root.left);
128   let right = this.preOrderTraversal(root.right);
129   return [root.val, ...left, ...right];
130 }
131
132 preOrderTraversalV2(root) {
133   if (!root) return [];
134   let newArray = new Array();

```

```

135     newArray.push(root.val);
136     newArray.push(...this.preOrderTraversalV2(root.left));
137     newArray.push(...this.preOrderTraversalV2(root.right));
138     return newArray;
139   }
140
141   inOrderTraversal(root) {
142     if (!root) return [];
143     let left = this.inOrderTraversal(root.left);
144     let right = this.inOrderTraversal(root.right);
145     return [...left, root.val, ...right];
146   }
147
148   inOrderTraversalV2(root) {
149     if (!root) return [];
150     let newArray = new Array();
151     newArray.push(...this.inOrderTraversalV2(root.left));
152     newArray.push(root.val);
153     newArray.push(...this.inOrderTraversalV2(root.right));
154     return newArray;
155   }
156
157   postOrderTraversal(root) {
158     if (!root) return [];
159     let left = this.postOrderTraversal(root.left);
160     let right = this.postOrderTraversal(root.right);
161     return [...left, ...right, root.val];
162   }
163
164   postOrderTraversalV2(root) {
165     if (!root) return [];
166     let newArray = new Array();
167     newArray.push(...this.postOrderTraversalV2(root.left));
168     newArray.push(...this.postOrderTraversalV2(root.right));
169     newArray.push(root.val);
170     return newArray;
171   }
172
173 // Iterative Depth First Search
174
175   iterativePreOrder(root) {
176     let stack = [root];
177     let results = [];
178     while (stack.length) {
179       let current = stack.pop();
180       results.push(current);
181       if (current.left) stack.push(current.left);
182       if (current.right) stack.push(current.right);
183     }
184     return results;
185   }
186
187   iterativeInOrder(root) {
188     let stack = [];
189     let current = root;
190     let results = [];
191     while (true) {
192       while (current) {
193         stack.push(current);
194         current = current.left;
195       }
196
197       if (!stack.length) break;
198       let removed = stack.pop();
199       results.push(removed);
200       current = removed.right;
201     }
202     return results;
203   }
204
205 //To-Do iterativePostOrder
206
207 //Breadth First Search
208
209   breadthFirstSearch(root) {
210     let queue = [root];

```

```
211 let result = [];
212 while (queue.length) {
213   let current = queue.shift();
214   if (current.left) queue.push(current.left);
215   if (current.right) queue.push(current.left);
216   current.push(result);
217 }
218 return result;
219 }
220
221 // Converting a Sorted Array to a Binary Search Tree
222
223 sortedArrayToBST(nums) {
224   if (nums.length === 0) return null;
225
226   let mid = Math.floor(nums.length / 2);
227   let root = new TreeNode(nums[mid]);
228
229   let left = nums.slice(0, mid);
230   root.left = sortedArrayToBST(left);
231
232   let right = nums.slice(mid + 1);
233   root.right = sortedArrayToBST(right);
234
235   return root;
236 }
237 }
```

BST Insert

```
1 class Node:
2     def __init__(self, val):
3         self.l_child = None
4         self.r_child = None
5         self.data = val
6
7     def binary_insert(self, node):
8         if root is None:
9             root = node
10        else:
11            if root.data > node.data:
12                if root.l_child is None:
13                    root.l_child = node
14                else:
15                    binary_insert(root.l_child, node)
16            else:
17                if root.r_child is None:
18                    root.r_child = node
19                else:
20                    binary_insert(root.r_child, node)
21
22     def in_order_print(self):
23         if not root:
24             return
25         in_order_print(root.l_child)
26         print root.data
27         in_order_print(root.r_child)
28
29     def pre_order_print(self):
30         if not root:
31             return
32         print root.data
33         pre_order_print(root.l_child)
34         pre_order_print(root.r_child)
```

```
1 r = Node(3)
2 binary_insert(r, Node(7))
3 binary_insert(r, Node(1))
4 binary_insert(r, Node(5))
```

```
1      3
2      / \
3      1   7
4          /
5          5
```

```
1 print "in order:"
2 in_order_print(r)
3
4 print "pre order"
5 pre_order_print(r)
6
7 in order:
8 1
9 3
10 5
11 7
12 pre order
13 3
```

14 1
15 7
16 5

Recursion

```
1 """Implement a function recursively to get the desired
2 Fibonacci sequence value.
3 Your code should have the same input/output as the
4 iterative code in the instructions."""
5
6 def get_fib(position):
7
8     output = 0
9     if(position==0):
10         return output
11
12     if(position==1):
13         return position
14     else:
15         output += get_fib(position-1)+get_fib(position-2)
16         return output
17
18 # Test cases
19 print get_fib(9)
20 print get_fib(11)
21 print get_fib(0)
22
```

Hash Table

```
1 """Write a HashTable class that stores strings
2 in a hash table, where keys are calculated
3 using the first two letters of the string."""
4
5 class HashTable(object):
6     def __init__(self):
7         self.table = [None]*10000
8
9     def store(self, string):
10        """Input a string that's stored in
11        the table."""
12        index = self.calculate_hash_value(string)
13        if(self.lookup(string)==-1):
14            self.table[index] = [string]
15        else:
16            self.table[index].append(string)
17        pass
18
19    def lookup(self, string):
20        """Return the hash value if the
21        string is already in the table.
22        Return -1 otherwise."""
23        index = self.calculate_hash_value(string)
24        if(self.table[index]!=None):
25            return index
26        else:
27            return -1
28
29    def calculate_hash_value(self, string):
30        """Helper function to calculate a
31        hash value from a string."""
32        hash_val = ord(string[0])*100+ord(string[1])
33        return hash_val
34
35 # Setup
36 hash_table = HashTable()
37
38 # Test calculate_hash_value
39 # Should be 8568
40 print hash_table.calculate_hash_value('UDACITY')
41
42 # Test lookup edge case
43 # Should be -1
44 print hash_table.lookup('UDACITY')
45
46 # Test store
47 hash_table.store('UDACITY')
48 # Should be 8568
49 print hash_table.lookup('UDACITY')
50
51 # Test store edge case
52 hash_table.store('UDACIOUS')
53 # Should be 8568
54 print hash_table.lookup('UDACIOUS')
55
56 #print(hash_table.table)
57
```

Linked List

Implementation

```
1  """The LinkedList code from before is provided below.
2  Add three functions to the LinkedList.
3  "get_position" returns the element at a certain position.
4  The "insert" function will add an element to a particular
5  spot in the list.
6  "delete" will delete the first element with that
7  particular value.
8  Then, use "Test Run" and "Submit" to run the test cases
9  at the bottom."""
10
11 class Element(object):
12     def __init__(self, value):
13         self.value = value
14         self.next = None
15
16 class LinkedList(object):
17     def __init__(self, head=None):
18         self.head = head
19
20     def append(self, new_element):
21         current = self.head
22         if self.head:
23             while current.next:
24                 current = current.next
25             current.next = new_element
26         else:
27             self.head = new_element
28
29     def get_position(self, position):
30         """Get an element from a particular position.
31         Assume the first position is "1".
32         Return "None" if position is not in the list."""
33         current = self.head
34         if(self.head):
35             for i in range(position)[1:]:
36                 if(current.next==None):
37                     return None
38                 else:
39                     current=current.next
40             return current
41         return None
42
43     def insert(self, new_element, position):
44         """Insert a new node at the given position.
45         Assume the first position is "1".
46         Inserting at position 3 means between
47         the 2nd and 3rd elements."""
48         current = self.head
49         if(self.head):
50             for i in range(position)[1:]:
51                 if(i==position-1):
52                     after = current.next
53                     current.next = new_element
54                     new_element.next = after
55                 elif(current.next!=None):
56                     current = current.next
57                 else:
58                     return 'position out of bounds'
59         pass
60
61
62     def delete(self, value):
63         """Delete the first node with a given value."""
64         current = self.head
65         if(self.head):
66             while(current.next!=None):
67                 if(current.next.value==value):
68                     after = current.next.next
69                     current.next = after
70                 else:
71                     current = current.next
```

```

73         if(self.head.value==value):
74             after = self.head.next
75             self.head = after
76             pass
77
78     # Test cases
79     # Set up some Elements
80     e1 = Element(1)
81     e2 = Element(2)
82     e3 = Element(3)
83     e4 = Element(4)
84
85     # Start setting up a LinkedList
86     ll = LinkedList(e1)
87     ll.append(e2)
88     ll.append(e3)
89
90     # Test get_position
91     # Should print 3
92     print ll.head.next.next.value
93     # Should also print 3
94     print ll.get_position(3).value
95
96     # Test insert
97     ll.insert(e4,3)
98     # Should print 4 now
99     print ll.get_position(3).value
100
101    # Test delete
102    ll.delete(1)
103    # Should print 2 now
104    print ll.get_position(1).value
105    # Should print 4 now
106    print ll.get_position(2).value
107    # Should print 3 now
108    print ll.get_position(3).value
109

```

Second Tab

A linked list is similar to an array, it holds values. However, links in a linked list do not have indexes.

- This is an example of a double ended, doubly linked list.
- Each link references the next link and the previous one.
- A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.
 - Advantages over SLL - It can be traversed in both forward and backward direction.
 - Delete operation is more efficient

```

1     """Each ListNode holds a reference to its previous node
2     as well as its next node in the List."""
3     class ListNode:
4         def __init__(self, value, prev=None, next=None):
5             self.value = value
6             self.prev = prev
7             self.next = next
8
9         """Wrap the given value in a ListNode and insert it
10        after this node. Note that this node could already
11        have a next node it is point to."""
12         def insert_after(self, value):
13             current_next = self.next

```

```

14     self.next = ListNode(value, self, current_next)
15     if current_next:
16         current_next.prev = self.next
17
18     """Wrap the given value in a ListNode and insert it
19     before this node. Note that this node could already
20     have a previous node it is point to."""
21     def insert_before(self, value):
22         current_prev = self.prev
23         self.prev = ListNode(value, current_prev, self)
24         if current_prev:
25             current_prev.next = self.prev
26
27     """Rearranges this ListNode's previous and next pointers
28     accordingly, effectively deleting this ListNode."""
29     def delete(self):
30         if self.prev:
31             self.prev.next = self.next
32         if self.next:
33             self.next.prev = self.prev
34
35     """Our doubly-linked list class. It holds references to
36     the list's head and tail nodes."""
37     class DoublyLinkedList:
38         def __init__(self, node=None):
39             self.head = node
40             self.tail = node
41             self.length = 1 if node is not None else 0
42
43         def __len__(self):
44             return self.length
45
46         def add_to_head(self, value):
47             pass
48
49         def remove_from_head(self):
50             pass
51
52         def add_to_tail(self, value):
53             pass
54
55         def remove_from_tail(self):
56             pass
57
58         def move_to_front(self, node):
59             pass
60
61         def move_to_end(self, node):
62             pass
63
64         def delete(self, node):
65             pass
66
67         def get_max(self):
68             pass
69

```

Test:

```

1 import unittest
2 from doubly_linked_list import ListNode
3 from doubly_linked_list import DoublyLinkedList
4
5 class DoublyLinkedListTests(unittest.TestCase):
6     def setUp(self):
7         self.node = ListNode(1)
8         self.dll = DoublyLinkedList(self.node)

```

```

9
10 def test_list_remove_from_tail(self):
11     self.dll.remove_from_tail()
12     self.assertIsNone(self.dll.head)
13     self.assertIsNone(self.dll.tail)
14     self.assertEqual(len(self.dll), 0)
15
16     self.dll.add_to_tail(33)
17     self.assertEqual(self.dll.head.value, 33)
18     self.assertEqual(self.dll.tail.value, 33)
19     self.assertEqual(len(self.dll), 1)
20     self.assertEqual(self.dll.remove_from_tail(), 33)
21     self.assertEqual(len(self.dll), 0)
22
23     self.dll.add_to_tail(68)
24     self.assertEqual(len(self.dll), 1)
25     self.assertEqual(self.dll.remove_from_tail(), 68)
26     self.assertEqual(len(self.dll), 0)
27
28 def test_list_remove_from_head(self):
29     self.dll.remove_from_head()
30     self.assertIsNone(self.dll.head)
31     self.assertIsNone(self.dll.tail)
32     self.assertEqual(len(self.dll), 0)
33
34     self.dll.add_to_head(2)
35     self.assertEqual(self.dll.head.value, 2)
36     self.assertEqual(self.dll.tail.value, 2)
37     self.assertEqual(len(self.dll), 1)
38     self.assertEqual(self.dll.remove_from_head(), 2)
39     self.assertEqual(len(self.dll), 0)
40
41     self.dll.add_to_head(55)
42     self.assertEqual(len(self.dll), 1)
43     self.assertEqual(self.dll.remove_from_head(), 55)
44     self.assertEqual(len(self.dll), 0)
45
46 def test_list_add_to_tail(self):
47     self.assertEqual(self.dll.tail.value, 1)
48     self.assertEqual(len(self.dll), 1)
49
50     self.dll.add_to_tail(30)
51     self.assertEqual(self.dll.tail.prev.value, 1)
52     self.assertEqual(self.dll.tail.value, 30)
53     self.assertEqual(len(self.dll), 2)
54
55     self.dll.add_to_tail(20)
56     self.assertEqual(self.dll.tail.prev.value, 30)
57     self.assertEqual(self.dll.tail.value, 20)
58     self.assertEqual(len(self.dll), 3)
59
60 def test_node_delete(self):
61     node_1 = ListNode(3)
62     node_2 = ListNode(4)
63     node_3 = ListNode(5)
64
65     node_1.next = node_2
66     node_2.next = node_3
67     node_2.prev = node_1
68     node_3.prev = node_2
69
70     node_2.delete()
71
72     self.assertEqual(node_1.next, node_3)
73     self.assertEqual(node_3.prev, node_1)
74
75 def test_node_insert_before(self):
76     self.node.insert_before(0)
77     self.assertEqual(self.node.prev.value, 0)
78
79 def test_list_add_to_head(self):
80     self.assertEqual(self.dll.head.value, 1)
81
82     self.dll.add_to_head(10)
83     self.assertEqual(self.dll.head.value, 10)
84     self.assertEqual(self.dll.head.next.value, 1)

```

```

85     self.assertEqual(len(self.dll), 2)
86
87 def test_node_insert_after(self):
88     self.node.insert_after(2)
89     self.assertEqual(self.node.next.value, 2)
90
91 def test_list_move_to_end(self):
92     self.dll.add_to_head(40)
93     self.assertEqual(self.dll.tail.value, 1)
94     self.assertEqual(self.dll.head.value, 40)
95
96     self.dll.move_to_end(self.dll.head)
97     self.assertEqual(self.dll.tail.value, 40)
98     self.assertEqual(self.dll.tail.prev.value, 1)
99     self.assertEqual(len(self.dll), 2)
100
101    self.dll.add_to_tail(4)
102    self.dll.move_to_end(self.dll.head.next)
103    self.assertEqual(self.dll.tail.value, 40)
104    self.assertEqual(self.dll.tail.prev.value, 4)
105    self.assertEqual(len(self.dll), 3)
106
107 def test_list_move_to_front(self):
108     self.dll.add_to_tail(3)
109     self.assertEqual(self.dll.head.value, 1)
110     self.assertEqual(self.dll.tail.value, 3)
111
112     self.dll.move_to_front(self.dll.tail)
113     self.assertEqual(self.dll.head.value, 3)
114     self.assertEqual(self.dll.head.next.value, 1)
115     self.assertEqual(len(self.dll), 2)
116
117     self.dll.add_to_head(29)
118     self.dll.move_to_front(self.dll.head.next)
119     self.assertEqual(self.dll.head.value, 3)
120     self.assertEqual(self.dll.head.next.value, 29)
121     self.assertEqual(len(self.dll), 3)
122
123 def test_list_delete(self):
124     self.dll.delete(self.node)
125     self.assertIsNone(self.dll.head)
126     self.assertIsNone(self.dll.tail)
127     self.assertEqual(len(self.dll), 0)
128
129     self.dll.add_to_tail(1)
130     self.dll.add_to_head(9)
131     self.dll.add_to_tail(6)
132
133     self.dll.delete(self.dll.head)
134     self.assertEqual(self.dll.head.value, 1)
135     self.assertEqual(self.dll.tail.value, 6)
136     self.assertEqual(len(self.dll), 2)
137
138     self.dll.delete(self.dll.head)
139     self.assertEqual(self.dll.head.value, 6)
140     self.assertEqual(self.dll.tail.value, 6)
141     self.assertEqual(len(self.dll), 1)
142
143 def test_get_max(self):
144     self.assertEqual(self.dll.get_max(), 1)
145     self.dll.add_to_tail(100)
146     self.assertEqual(self.dll.get_max(), 100)
147     self.dll.add_to_tail(55)
148     self.assertEqual(self.dll.get_max(), 100)
149     self.dll.add_to_tail(101)
150     self.assertEqual(self.dll.get_max(), 101)
151
152 if __name__ == '__main__':
153     unittest.main()

```

Double Linked List

```
1 # Each ListNode holds a reference to its previous node
2 # as well as its next node in the List.
3
4
5 class ListNode:
6     def __init__(self, value, prev=None, next=None):
7         self.value = value
8         self.prev = prev
9         self.next = next
10
11    def __repr__(self):
12        return (
13            "Value: {}, ".format(self.value if self.value else None)
14            + "Prev: {}, ".format(self.prev.value if self.prev else None)
15            + "Next: {} \n".format(self.next.value if self.next else None)
16        )
17    # Wrap the given value in a ListNode and insert it
18    # after this node. Note that this node could already
19    # have a next node it is pointing to.
20
21    def insert_after(self, value):
22        current_next = self.next
23        self.next = ListNode(value, self, current_next)
24        if current_next:
25            current_next.prev = self.next
26    # Wrap the given value in a ListNode and insert it
27    # before this node. Note that this node could already
28    # have a previous node it is pointing to.
29
30    def insert_before(self, value):
31        current_prev = self.prev
32        self.prev = ListNode(value, current_prev, self)
33        if current_prev:
34            current_prev.next = self.prev
35    # Rearranges this ListNode's previous and next pointers
36    # accordingly, effectively deleting this ListNode.
37
38    def delete(self):
39        if self.prev:
40            self.prev.next = self.next
41        if self.next:
42            self.next.prev = self.prev
43    # Our doubly-linked list class. It holds references to
44    # the list's head and tail nodes.
45
46
47 class DoublyLinkedList:
48     def __init__(self, node=None):
49         self.head = node
50         self.tail = node
51         self.length = 1 if node is not None else 0
52
53     def __repr__(self):
54         return f"Head: {self.head} \n Tail: {self.tail} \n Length: {self.length}"
55
56     def __len__(self):
57         return self.length
58    # Replaces the head of the list with a new value that is passed in.
59
60    def add_to_head(self, value):
61        new_node = ListNode(value)
62        self.length += 1
63    # if there is no head or tail, it needs to become both:
```

```

64         if not self.head and not self.tail:
65             self.head = new_node
66             self.tail = new_node
67 # otherwise it only needs to become the head:
68     else:
69         self.head.prev = new_node
70         new_node.next = self.head
71         self.head = new_node
72 # Replaces the tail of the list with a new value that is passed in.
73
74     def remove_from_head(self):
75         # if there is no head, just return None because we can't remove
76         if not self.head:
77             return None
78 # reduce the length
79         self.length -= 1
80 # we need to store the current head to return it once removed
81         current_head = self.head
82 # if there is solely one node, we set both head and tail to None
83         if self.head == self.tail:
84             self.head = None
85             self.tail = None
86             return current_head.value
87 # changes the head to the next node
88     else:
89         self.head = self.head.next
90 # Removes pointers to any previous node
91         self.head.prev = None
92         return current_head.value
93 # Removes the head node and returns the value stored in it.
94
95     def add_to_tail(self, value):
96         new_node = ListNode(value)
97         self.length += 1
98 # if there is no head or tail, it needs to become both:
99         if not self.head and not self.tail:
100             self.head = new_node
101             self.tail = new_node
102 # otherwise it only needs to become the tail:
103     else:
104         self.tail.next = new_node
105         new_node.prev = self.tail
106         self.tail = new_node
107 # Removes the tail node and returns the value stored in it
108
109     def remove_from_tail(self):
110         # if there is no tail, just return None because we can't remove
111         if not self.tail:
112             return None
113 # reduce the length
114         self.length -= 1
115 # we need to store the current tail to return it once removed
116         current_tail = self.tail
117 # if there is solely one node, we set both head and tail to None
118         if self.head == self.tail:
119             self.head = None
120             self.tail = None
121             return current_tail.value
122 # changes the tail to the next node
123     else:
124         self.tail = self.tail.prev
125 # Removes pointers to any next node
126         self.tail.next = None
127         return current_tail.value
128 # Takes a reference to a node in the list and moves it to the front of the list, shifting all other list nodes down
129
130     def move_to_head(self, node):
131         # if the passed node is already the head, we just return it
132         if node is self.head:
133             return node
134 # if the passed node is the tail, we need to remove it from the tail
135         if node is self.tail:
136             self.remove_from_tail()
137         else:
138             node.delete()
139             self.length -= 1

```

```

140 # we should add it but only the value of the passed node
141         self.add_to_head(node.value)
142 # Takes a reference to a node in the list and moves it to the end of the list, shifting all other list nodes up.
143
144     def move_to_tail(self, node):
145         if node is self.tail:
146             return node
147         if node is self.head:
148             self.remove_from_head()
149         else:
150             node.delete()
151             self.length -= 1
152             self.add_to_tail(node.value)
153 # Takes a reference to a node in the list and removes it from the list. The deleted node's `previous` and `next` p
154
155     def delete(self, node):
156         if self.head is self.tail:
157             self.remove_from_head()
158         elif node is self.head:
159             self.remove_from_head()
160         elif node is self.tail:
161             self.remove_from_tail()
162         else:
163             node.delete()
164             return node.value
165 # Returns the maximum value in the list.
166
167     def get_max(self):
168         # if there is no head, we know the list is empty
169         if not self.head:
170             return None
171 # we'll set our starting max value as the first value we'll begin looping through in the list, the head
172         max_value = self.head.value
173 # we'll set a current value to check against
174         current = self.head
175         while current:
176             if current.value > max_value:
177                 max_value = current.value
178         # increment current
179         current = current.next
180 # once all values are checked, return max value
181         return max_value
182
183
184 ll = DoublyLinkedList()
185 print(f"ll: {ll}") # should be empty
186 ll.add_to_head(2) # should be 2
187 ll.add_to_head(5) # should be 5,2
188 ll.add_to_head(7) # should be 7,5,2
189 ll.remove_from_head() # should be 5,2
190 ll.add_to_tail(9) # should be 5,2,9
191 ll.add_to_tail(11) # should be 5,2,9,11
192 ll.add_to_tail(13) # should be 5,2,9,11,13
193 ll.remove_from_tail() # should be 5,2,9,11
194 ll.get_max() # should return 11
195 print(f"ll: {ll}") # return length 4, head: 5, tail: 11
196

```

Sorting

```
1 def partition(A, lo, hi):
2     pivot = A[lo + (hi - lo) // 2]
3     i = lo - 1
4     j = hi + 1
5
6     while True:
7
8         i += 1
9         while A[i] < pivot:
10             i += 1
11
12         j -= 1
13         while A[j] > pivot:
14             j -= 1
15
16         if i >= j:
17             return j
18         A[i], A[j] = A[j], A[i]
19
20
21 def quicksort(A, lo, hi):
22     if lo < hi:
23         p = partition(A, lo, hi)
24         quicksort(A, lo, p)
25         quicksort(A, p + 1, hi)
26     return A
27
28
29 if __name__ == "__main__":
30     arr = [8, 3, 5, 1, 7, 2]
31     quicksort(arr, 0, len(arr) - 1)
32     # >>> [1, 2, 3, 5, 7, 8]
33
```

Bubble Sort

Script Name

Bubble Sort Algorithm.

Aim

To write a program for Bubble sort.

Purpose

To get a understanding about Bubble sort.

Short description of package/script

- It is a python program of Bubble sort Algorithm.
- It is written in a way that it takes user input.

Workflow of the Project

- First a function is written to perform Bubble sort.
- Then outside the function user input is taken.

Detailed explanation of script, if needed

Start with the first element, compare the current element with the next element of the array. If the current element is greater than the next element of the array, swap both of them. If the current element is less than the next element, move to the next element. Keep on comparing the current element with all the elements in the array. The largest element of the array comes to its original position after 1st iteration. Repeat all the steps till the array is sorted.

Example

```

1 Consider an array a=[5,4,3,2,1]
2 Iteration 1:-
3      |5|4|3|2|1|
4      |_____5>4 therefore we swap both of them.
5      |4|5|3|2|1|
6      |_____5>3 therefore we swap both.
7      |4|3|5|2|1|
8      |_____5>2 therefore we swap.
9      |4|3|2|5|1|
10     |_____5>1 therefore we swap.
11     |4|3|2|1|5| Now 5 is placed at its original position
12
13 Iteration 2:-
14     |4|3|2|1|5|
15     |_____4>3 therefore we swap both.
16     |3|4|2|1|5|
17     |_____4>2 therefore we swap both.
18     |3|2|4|1|5|
19     |_____4>1 therefore we swap both.
20     |3|2|1|4|5|
21     |__ 4 is placed at its original position.
22
23 Iteration 3:-
24     |3|2|1|4|5|
25     |_____3>2 we swap.
26     |2|3|1|4|5|
27     |_____3>1 we swap.
28     |2|1|3|4|5|- 3 is placed at original position.
29
30 Iteration 4:-
31     |2|1|3|4|5|
32     |_____2>1 we swap.
33     |1|2|3|4|5| the array is sorted.

```

Setup instructions

Just clone the repository .

Output

```

1 #Link to problem:-
2 #Bubble sort is a sorting algorithm. Sorting algorithms are used to arrange the array in particular order.In,Bubble

```

```

3
4 def bubbleSort(a):
5     n = len(a)
6     # Traverse through all array elements
7
8     for i in range(n-1):
9         # Last i elements are already in place
10        for j in range(0, n-i-1):
11
12            # traverse the array from 0 to n-i-1
13            # Swap if the element found is greater
14            # than the next element
15            if arr[j] > arr[j + 1] :
16                arr[j], arr[j + 1] = arr[j + 1], arr[j]
17
18 arr = []
19 n=int(input("Enter size of array: "))
20 for i in range(n):
21     e=int(input())
22     arr.append(e)
23 bubbleSort(arr)
24 print ("Sorted array is:")
25 for i in range(len(arr)):
26     print(arr[i])
27
28 #Time complexity - O(n^2)
29 #Space complexity - O(1)

```

Insertion Sort

```

1 # Insertion Sort
2
3 ## Aim
4
5 The main aim of the script is to sort numbers in list using insertion sort.
6
7 ## Purpose
8
9 The main purpose is to sort list of any numbers in O(n) or O(n^2) time complexity.
10
11 ## Short description of package/script
12
13 Takes in an array. <br>
14 Sorts the array and prints sorted array along with the number of swaps and comparisions made.
15 Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. T
16
17 ## Detailed explanation of script, if needed
18
19 To sort an array of size n in ascending order: <br>
20 1: Iterate from a[1] to a[n] over the array. <br>
21 2: Compare the current element (val) to its predecessor. <br>
22 3: If the val is smaller than its predecessor, compare it to the elements before. Move the greater elements one pos
23
24 ## Setup instructions
25
26 Download code and run it in any python editor. Latest version is always better.
27
28 ## Compilation Steps
29

```

```

30 1>Edit array a and enter your array/list you want to sort. 2. Run the code
31
32 ## Sample Test Cases
33
34 ### Test case 1:
35
36 input:<br>
37 a = [34,5,77,33] <br>
38
39 output :<br>
40
41 5, 33, 34, 77 along with <br>
42 no. of swaps = 3 <br>
43 no. of comparisons=5<br>
44
45 ### Test case 2
46
47 input<br>
48 a=[90,8,11,3,2000,700,478] <br>
49
50 Output:<br>
51
52 No. of swaps= 8 <br>
53 No. of comparisions=12 <br>
54 Sorted Array is: <br>
55 3 8 11 90 478 700 2000<br>
56
57 ### Test case 3
58
59 input<br>
60 a=[0,33,7000,344,-88,2000]<br>
61
62 output:<br>
63
64 No. of swaps= 6<br>
65 No. of comparisions=10<br>
66 Sorted Array is:<br>
67 -88 0 33 344 2000 7000<br>
68
69 ## Output
70
71 
72 
73
74 ## Author(s)
75
76 [Ritika Chand](https://github.com/RC2208)
77
78

```

Searching

Graphs

Directed Graph:

Directed Graph:

```
1  class Node(object):
2      def __init__(self, value):
3          self.value = value
4          self.edges = []
5
6  class Edge(object):
7      def __init__(self, value, node_from, node_to):
8          self.value = value
9          self.node_from = node_from
10         self.node_to = node_to
11
12 class Graph(object):
13     def __init__(self, nodes=[], edges=[]):
14         self.nodes = nodes
15         self.edges = edges
16
17     def insert_node(self, new_node_val):
18         new_node = Node(new_node_val)
19         self.nodes.append(new_node)
20
21     def insert_edge(self, new_edge_val, node_from_val, node_to_val):
22         from_found = None
23         to_found = None
24         for node in self.nodes:
25             if node_from_val == node.value:
26                 from_found = node
27             if node_to_val == node.value:
28                 to_found = node
29         if from_found == None:
30             from_found = Node(node_from_val)
31             self.nodes.append(from_found)
32         if to_found == None:
33             to_found = Node(node_to_val)
34             self.nodes.append(to_found)
35         new_edge = Edge(new_edge_val, from_found, to_found)
36         from_found.edges.append(new_edge)
37         to_found.edges.append(new_edge)
38         self.edges.append(new_edge)
39
40     def get_edge_list(self):
41         """Don't return a list of edge objects!
42         Return a list of triples that looks like this:
43         (Edge Value, From Node Value, To Node Value)"""
44         edge_list = []
45         for item in self.nodes:
46             for each_edge in item.edges:
47                 edge_tuple = (each_edge.value, each_edge.node_from.value, each_edge.node_to.value)
48                 if edge_tuple not in edge_list:
49                     edge_list.append(edge_tuple)
50         return edge_list
51
52     def get_adjacency_list(self):
53         """Don't return any Node or Edge objects!
54         You'll return a list of lists.
55         The indices of the outer list represent
56         "from" nodes.
57         Each section in the list will store a list
58         of tuples that looks like this:
59         (To Node, Edge Value)"""
60         adjacency_list = []
61         edge_list = self.get_edge_list()
62         max_node_val = 0
63         for tuple in edge_list:
```

```

64         if(max_node_val<tuple[1]):
65             max_node_val = tuple[1]
66         if(max_node_val<tuple[2]):
67             max_node_val = tuple[2]
68
69     for i in range(max_node_val+1):
70         node_list = []
71         for item in edge_list:
72             if(i==item[1]):
73                 new_tuple = (item[2], item[0])
74                 node_list.append(new_tuple)
75         if(node_list!=[]):
76             adjacency_list.append(node_list)
77         else:
78             adjacency_list.append(None)
79
80     return adjacency_list
81
82 def get_adjacency_matrix(self):
83     """Return a matrix, or 2D list.
84     Row numbers represent from nodes,
85     column numbers represent to nodes.
86     Store the edge values in each spot,
87     and a 0 if no edge exists."""
88     adjacency_matrix = []
89     adjacency_list = self.get_adjacency_list()
90     max_node_val = len(adjacency_list)
91
92     for item in adjacency_list:
93         node_list = [0]*(max_node_val)
94         if(item!=None):
95             for tuple in item:
96                 node_list[tuple[0]]=tuple[1]
97         adjacency_matrix.append(node_list)
98     return adjacency_matrix
99
100 graph = Graph()
101 graph.insert_edge(100, 1, 2)
102 graph.insert_edge(101, 1, 3)
103 graph.insert_edge(102, 1, 4)
104 graph.insert_edge(103, 3, 4)
105 # Should be [(100, 1, 2), (101, 1, 3), (102, 1, 4), (103, 3, 4)]
106 print(graph.get_edge_list())
107 # Should be [None, [(2, 100), (3, 101), (4, 102)], None, [(4, 103)], None]
108 print(graph.get_adjacency_list())
109 # Should be [[0, 0, 0, 0, 0], [0, 0, 100, 101, 102], [0, 0, 0, 0, 0], [0, 0, 0, 0, 103], [0, 0, 0, 0, 0]]
110 print(graph.get_adjacency_matrix())

```

Graph Traversal

```
 1  class Node(object):
 2      def __init__(self, value):
 3          self.value = value
 4          self.edges = []
 5          self.visited = False
 6
 7  class Edge(object):
 8      def __init__(self, value, node_from, node_to):
 9          self.value = value
10          self.node_from = node_from
11          self.node_to = node_to
12
13 # You only need to change code with docs strings that have TODO.
14 # Specifically: Graph.dfs_helper and Graph.bfs
15 # New methods have been added to associate node numbers with names
16 # Specifically: Graph.set_node_names
17 # and the methods ending in "_names" which will print names instead
18 # of node numbers
19
20 class Graph(object):
21     def __init__(self, nodes=None, edges=None):
22         self.nodes = nodes or []
23         self.edges = edges or []
24         self.node_names = []
25         self._node_map = {}
26
27     def set_node_names(self, names):
28         """The Nth name in names should correspond to node number N.
29         Node numbers are 0 based (starting at 0).
30         """
31         self.node_names = list(names)
32
33     def insert_node(self, new_node_val):
34         """Insert a new node with value new_node_val"""
35         new_node = Node(new_node_val)
36         self.nodes.append(new_node)
37         self._node_map[new_node_val] = new_node
38         return new_node
39
40     def insert_edge(self, new_edge_val, node_from_val, node_to_val):
41         """Insert a new edge, creating new nodes if necessary"""
42         nodes = {node_from_val: None, node_to_val: None}
43         for node in self.nodes:
44             if node.value in nodes:
45                 nodes[node.value] = node
46             if all(nodes.values()):
47                 break
48         for node_val in nodes:
49             nodes[node_val] = nodes[node_val] or self.insert_node(node_val)
50         node_from = nodes[node_from_val]
51         node_to = nodes[node_to_val]
52         new_edge = Edge(new_edge_val, node_from, node_to)
53         node_from.edges.append(new_edge)
54         node_to.edges.append(new_edge)
55         self.edges.append(new_edge)
56
57     def get_edge_list(self):
58         """Return a list of triples that looks like this:
59         (Edge Value, From Node, To Node)"""
60         return [(e.value, e.node_from.value, e.node_to.value)
61                 for e in self.edges]
62
63     def get_edge_list_names(self):
64         """Return a list of triples that looks like this:
65         (Edge Value, From Node Name, To Node Name)"""
66         return [(edge.value,
67                  self.node_names[edge.node_from.value],
68                  self.node_names[edge.node_to.value])
69                 for edge in self.edges]
70
71     def get_adjacency_list(self):
72         """Return a list of lists.
```

```

73     The indecies of the outer list represent "from" nodes.
74     Each section in the list will store a list
75     of tuples that looks like this:
76     (To Node, Edge Value)"""
77     max_index = self.find_max_index()
78     adjacency_list = [[] for _ in range(max_index)]
79     for edg in self.edges:
80         from_value, to_value = edg.node_from.value, edg.node_to.value
81         adjacency_list[from_value].append((to_value, edg.value))
82     return [a or None for a in adjacency_list] # replace []'s with None
83
84 def get_adjacency_list_names(self):
85     """Each section in the list will store a list
86     of tuples that looks like this:
87     (To Node Name, Edge Value).
88     Node names should come from the names set
89     with set_node_names."""
90     adjacency_list = self.get_adjacency_list()
91     def convert_to_names(pair, graph=self):
92         node_number, value = pair
93         return (graph.node_names[node_number], value)
94     def map_conversion(adjacency_list_for_node):
95         if adjacency_list_for_node is None:
96             return None
97         return map(convert_to_names, adjacency_list_for_node)
98     return [map_conversion(adjacency_list_for_node)
99            for adjacency_list_for_node in adjacency_list]
100
101 def get_adjacency_matrix(self):
102     """Return a matrix, or 2D list.
103     Row numbers represent from nodes,
104     column numbers represent to nodes.
105     Store the edge values in each spot,
106     and a 0 if no edge exists."""
107     max_index = self.find_max_index()
108     adjacency_matrix = [[0] * (max_index) for _ in range(max_index)]
109     for edg in self.edges:
110         from_index, to_index = edg.node_from.value, edg.node_to.value
111         adjacency_matrix[from_index][to_index] = edg.value
112     return adjacency_matrix
113
114 def find_max_index(self):
115     """Return the highest found node number
116     Or the length of the node names if set with set_node_names()."""
117     if len(self.node_names) > 0:
118         return len(self.node_names)
119     max_index = -1
120     if len(self.nodes):
121         for node in self.nodes:
122             if node.value > max_index:
123                 max_index = node.value
124     return max_index
125
126 def find_node(self, node_number):
127     "Return the node with value node_number or None"
128     return self._node_map.get(node_number)
129
130 def _clear_visited(self):
131     for node in self.nodes:
132         node.visited = False
133
134 def dfs_helper(self, start_node):
135     """TODO: Write the helper function for a recursive implementation
136     of Depth First Search iterating through a node's edges. The
137     output should be a list of numbers corresponding to the
138     values of the traversed nodes.
139     ARGUMENTS: start_node is the starting Node
140     MODIFIES: the value of the visited property of nodes in self.nodes
141     RETURN: a list of the traversed node values (integers).
142     """
143     ret_list = [start_node.value]
144     # Your code here
145     start_node.visited = True
146     next_edges = start_node.edges
147     next_vals = []
148     for this_edge in next_edges:

```

```

149         #iterates through edges to find the first edge that has places not 'visited'
150         if(this_edge.node_to.visited==False):
151             # adds all places not visited along this edge to the list of nodes
152             # to be returned, using recursion
153             next_vals.extend(self.dfs_helper(this_edge.node_to))
154
155         if(next_vals!=[]):
156             ret_list.extend(next_vals)
157
158     return ret_list
159
160 def dfs(self, start_node_num):
161     """Outputs a list of numbers corresponding to the traversed nodes
162     in a Depth First Search.
163     ARGUMENTS: start_node_num is the starting node number (integer)
164     MODIFIES: the value of the visited property of nodes in self.nodes
165     RETURN: a list of the node values (integers)."""
166     self._clear_visited()
167     start_node = self.find_node(start_node_num)
168     return self.dfs_helper(start_node)
169
170 def dfs_names(self, start_node_num):
171     """Return the results of dfs with numbers converted to names."""
172     return [self.node_names[num] for num in self.dfs(start_node_num)]
173
174 def bfs(self, start_node_num):
175     """TODO: Create an iterative implementation of Breadth First Search
176     iterating through a node's edges. The output should be a list of
177     numbers corresponding to the traversed nodes.
178     ARGUMENTS: start_node_num is the node number (integer)
179     MODIFIES: the value of the visited property of nodes in self.nodes
180     RETURN: a list of the node values (integers)."""
181     node = self.find_node(start_node_num)
182     self._clear_visited()
183     ret_list = [node.value]
184     # Your code here
185     node.visited = True
186     total_edges = node.edges
187     node_queue = [node]
188
189     while(node_queue!=[]):
190         this_node = node_queue[0]
191         node_queue = node_queue[1:]
192         node_edges = this_node.edges
193         for each_edge in node_edges:
194             if(each_edge.node_to.visited==False):
195                 node_queue.append(each_edge.node_to)
196                 each_edge.node_to.visited = True
197                 ret_list.append(each_edge.node_to.value)
198
199     return ret_list
200
201 def bfs_names(self, start_node_num):
202     """Return the results of bfs with numbers converted to names."""
203     return [self.node_names[num] for num in self.bfs(start_node_num)]
204
205 graph = Graph()
206
207 # You do not need to change anything below this line.
208 # You only need to implement Graph.dfs_helper and Graph.bfs
209
210 graph.set_node_names(('Mountain View',      # 0
211                       'San Francisco',   # 1
212                       'London',          # 2
213                       'Shanghai',        # 3
214                       'Berlin',          # 4
215                       'Sao Paolo',       # 5
216                       'Bangalore'))      # 6
217
218 graph.insert_edge(51, 0, 1)      # MV <-> SF
219 graph.insert_edge(51, 1, 0)      # SF <-> MV
220 graph.insert_edge(9950, 0, 3)    # MV <-> Shanghai
221 graph.insert_edge(9950, 3, 0)    # Shanghai <-> MV
222 graph.insert_edge(10375, 0, 5)   # MV <-> Sao Paolo
223 graph.insert_edge(10375, 5, 0)   # Sao Paolo <-> MV
224 graph.insert_edge(9900, 1, 3)    # SF <-> Shanghai
225 graph.insert_edge(9900, 3, 1)    # Shanghai <-> SF

```

```

225 graph.insert_edge(9130, 1, 4)    # SF <-> Berlin
226 graph.insert_edge(9130, 4, 1)    # Berlin <-> SF
227 graph.insert_edge(9217, 2, 3)    # London <-> Shanghai
228 graph.insert_edge(9217, 3, 2)    # Shanghai <-> London
229 graph.insert_edge(932, 2, 4)     # London <-> Berlin
230 graph.insert_edge(932, 4, 2)     # Berlin <-> London
231 graph.insert_edge(9471, 2, 5)    # London <-> Sao Paolo
232 graph.insert_edge(9471, 5, 2)    # Sao Paolo <-> London
233 # (6) 'Bangalore' is intentionally disconnected (no edges)
234 # for this problem and should produce None in the
235 # Adjacency List, etc.
236
237 import pprint
238 pp = pprint.PrettyPrinter(indent=2)
239
240 print("Edge List")
241 pp.pprint(graph.get_edge_list_names())
242
243 print("\nAdjacency List")
244 pp.pprint(graph.get_adjacency_list_names())
245
246 print("\nAdjacency Matrix")
247 pp.pprint(graph.get_adjacency_matrix())
248
249 print("\nDepth First Search")
250 pp.pprint(graph.dfs_names(2))
251
252 # Should print:
253 # Depth First Search
254 # ['London', 'Shanghai', 'Mountain View', 'San Francisco', 'Berlin', 'Sao Paolo']
255
256 print("\nBreadth First Search")
257 pp.pprint(graph.bfs_names(2))
258 # test error reporting
259 # pp.pprint(['Sao Paolo', 'Mountain View', 'San Francisco', 'London', 'Shanghai', 'Berlin'])
260
261 # Should print:
262 # Breadth First Search
263 # ['London', 'Shanghai', 'Berlin', 'Sao Paolo', 'Mountain View', 'San Francisco']
264

```

Exotic

data-structures

Main Categories Of Abstract Data Structures & Algorithms

Binary Search

Binary Search Tree

Stack

Queue

Linked List

Untitled

Untitled

Untitled

Untitled

Untitled

Resources

Python VS JavaScript

Javascript Python cheatsheet

Tom Tarpey edited this page on May 26, 2019 · 24 revisions

Contents

- Versions
- Development Environments
- Running Programs
- Comments
- Semicolons
- Whitespace, Blocks
- Functions
- Arithmetic Operators
- Variables
- Data Types
- Arrays/Lists
- Slices
- Objects/Dicts
- String Formatting
- Booleans and Conditionals
- `for` Loops
- `while` Loops
- `switch` Statement
- `if` Conditionals
- Classes

Versions

JavaScript

The standard defining JavaScript (JS) is *ECMAScript* (ES). Modern browsers and NodeJS support ES6, which has a rich feature set. Older browsers might not support all ES6 features.

The website [caniuse.com](#) will show which browsers support specific JS features.

Python

Python 3.x is the current version, but there are a number of packages and sites running legacy Python 2.

On some systems, you might have to be explicit when you invoke Python about which version you want by running `python2` or `python3`. The `--version` command line switch will tell you which version is running. Example:

```
1 $ python --version
2 Python 2.7.10
3 $ python2 --version
4 -bash: python2: command not found
5 $ python3 --version
6 Python 3.6.5
```

Using `virtualenv` or `pipenv` can really ease development painpoints surrounding the version. See [Development Environments](#), below.

Development Environments

JavaScript

For managing project packages, the classic tool is `npm`. This is slowly being superseded by the newer `yarn` tool. Choose one for a project, and don't mix and match.

Python

For managing project packages and Python versions, the classic tool is `virtualenv`. This is slowly being superseded by the newer `pipenv` tool.

Running Programs

JavaScript

Running from the command line with NodeJS:

```
node program.js arg1 arg2 etc
```

In a web page, a script is referenced with a `<script>` HTML tag:

```
<script src="program.js"></script>
```

Python

Running from the command line:

```
python program.py arg1 arg2 etc
```

Comments

JavaScript

Single line:

```
1 // Anything after two forward slashes is a comment
2 print(2); // prints 2
```

Multi-line comments:

```
1 /* Anything between slash-star and
2 star-slash is a comment */
```

You may not nest multi-line comments.

Python

Single line:

```
1 # Anything after a # is a comment
2 print(2) # prints 2
```

Python doesn't directly support multi-line comments, but you can effectively do them by using multi-line strings `"""`:

```
1 """
2 At this point we wish
3 to print out some numbers
4 and see where that gets us
5 """
6 print(1)
7 print(2)
```

Semicolons

JavaScript

Javascript ends statements with semicolons, usually at the end of the line. I can also be effectively used to put multiple statements on the same line, but this is rare.

```
1 console.log("Hello, world!");
2
3 let x = 10; console.log(x);
```

Javascript interpreters will let you get away without using semicolons at ends of lines, but you should use them.

Python

Python can separate statements by semicolons, though this is rare in practice.

```
print(1); print(2) # prints 1, then 2
```

Whitespace, blocks

JavaScript

Whitespace has no special meaning. Blocks are declared with squirrely braces `{` and `}`.

```
1 if (x == 2) {
2   console.log("x must be 2")
3 } else {
4   if (x == 3) {
5     console.log("x must be 3")
6   } else {
7     console.log("x must be something else")
8   }
9 }
```

Python

Indentation level is how blocks are declared. The preferred method is to use spaces, but tabs can also be used.

```
1 if x == 2:
2     print("x must be 2")
3 else:
4     if x == 3:
5         print("x must be 3")
6     else:
7         print("x must be something else")
```

Functions

JavaScript

Define functions as follows:

```
1 function foobar(x, y, z) {
2     console.log(x, y, z);
3     return 12;
4 }
```

An alternate syntax for functions is growing increasingly common, called *arrow functions*:

```
1 let hello = () => {
2     console.log("hello");
3     console.log("world");
4 }
5
6 hello(); // prints hello, then world
7
8 // Arrow functions with single parameters don't
9 // need parens around the parameter:
10 let printNum = x => {
11     console.log(x);
12 }
13
14 // If you don't explicitly return a value, the value
15 // of the last expression in the function is used.
16 // Also, single-expression functions don't need
17 // braces around them.
18 let add = (x, y) => x + y;
19
20 console.log(add(4, 5)); // prints 9
```

Python

Define functions as follows:

```
1 def foobar(x, y, z):
2     print(x, y, z)
3     return 12
```

Python also supports the concept of *lambda functions*, which are simple functions that can do basic operations.

```
1 add = lambda x, y: x + y
2
3 print(add(4, 5)) # prints 9
```

Arithmetic Operators

JavaScript

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
--	Pre-decrement, post-decrement
++	Pre-increment, post-increment
**	Exponentiation (power)
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulo assignment

Python

The pre- and post-increment and decrement are notably absent.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
**	Exponentiation (power)
=	Assignment
+=	Addition assignment

--	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulo assignment

Variables

Javascript

Variables are created upon use, but should be created with the `let` or `const` keywords.

```
1 let x = 10;
2 const y = 30;
```

`var` is an outdated way of declaring variables in Javascript.

Python

Variables are created upon use.

```
x = 10
```

Data Types

JavaScript

```
1 let a = 12;           // number
2 let b = 1.2;          // number
3 let c = 'hello';     // string
4 let d = "world";     // string
5 let e = true;         // boolean
6 let f = null;         // null value
7 let g = undefined;   // undefined value
```

Multi-line strings:

```
1 let s = `this is a
2 multi-line string`;
```

Parameterized strings:

```
1 let x = 12;
2 console.log(`x is ${x}`); // prints "x is 12"
```

JS is *weakly typed* so it supports operations on multiple types of data at once.

```
1 "2" + 4;           // string "24"
2 parseInt("2") + 4; // number 6
3 Number("2") + 4;  // number 6
```

Python

```
1 a = 12      # int (integer)
2 b = 1.2     # float (floating point)
3 c = 'hello' # str (string)
4 d = "world" # str
5 e = False   # bool (boolean)
6 f = None    # null value
```

Multi-line strings:

```
1 s = """this is a
2 multi-line string"""
```

Parameterized strings:

```
1 x = 12
2 print(f'x is {x}')  # prints "x is 12"
```

Python is generally *strongly typed* so it will often complain if you try to mix and match types. You can coerce a type with the `int()` , `float()` , `str()` , and `bool()` functions.

```
1 "2" + 4      # ERROR: can't mix types
2 int("2") + 4 # integer 6
3 "2" + str(4) # string 24
```

Arrays/Lists

JavaScript

In JS, lists are called *arrays*.

Arrays are zero-based.

Creating lists:

```
1 let a1 = new Array();  // Empty array
2 let a2 = new Array(10); // Array of 10 elements
3 let a3 = [] ;          // Empty array
4 let a4 = [10, 20, 30]; // Array of 3 elements
5 let a5 = [1, 2, "b"] ; // No problem
```

Accessing:

```
1 console.log(a4[1]); // prints 20
2
3 a4[0] = 5; // change from 10 to 5
4 a4[20] = 99; // OK, makes a new element at index 20
```

Length/number of elements:

```
a4.length; // 3
```

Python

In Python, arrays are called *lists*.

Lists are zero-based.

Creating lists:

```
1 a1 = list()          # Empty list
2 a2 = list((88, 99)) # List of two elements
3 a3 = []              # Empty list
4 a4 = [10, 20, 30]   # List of 3 elements
5 a5 = [1, 2, "b"]    # No problem
```

Accessing:

```
1 print(a4[1]) # prints 20
2
3 a4[0] = 5;    # change from 10 to 5
4 a4[20] = 99; # ERROR: assignment out of range
```

Length/Number of elements:

```
len(a4) # 3
```

Slices

In Python, we can access parts of lists or strings using slices.

Creating slices:

```
1 a[start:end] # items start through end-1
2 a[start:]    # items start through the rest of the array
3 a[:end]      # items from the beginning through end-1
4 a[:]         # a copy of the whole array
```

Starting from the end: We can also use negative numbers when creating slices, which just means we start with the index at the end of the array, rather than the index at the beginning of the array.

```
1 a[-1]    # last item in the array
2 a[-2:]   # last two items in the array
3 a[:-2]   # everything except the last two items
```

Tuples

Python supports a read-only type of list called a *tuple*.

```
1 x = (1, 2, 3)
2 print(x[1])  # prints 2
3
4 y = (10,)    # A tuple of one element, comma required
```

List Comprehensions

Python supports building lists with *list comprehensions*. This is often useful for filtering lists.

```
1 a = [1, 2, 3, 4, 5]
2
3 # Make a list b that is the same as list a:
4 b = [i for i in a]  # Pretty boring
5
6 # Make a list c that contains only the
7 # even elements of a:
8 c = [i for i in a if i % 2 == 0]
```

Objects/Dicts

JavaScript

Objects hold data which can be found by a specific key called a *property*.

Creation:

```
1 let o1 = {};           // empty object
2 let o2 = {"x": 12};    // one property
3 let o3 = {y: "hello"}; // property quotes optional
4
5 let o4 = { // common multiline format
6   "a": 20,
7   "b": 1.2,
8   "foo": "hello"
9 };
```

Access:

```
1 console.log(o2.x);    // prints 12
2 console.log(o4["foo"]); // prints hello
```

Python

Dicts hold information that can be accessed by a *key*.

Unlike objects in JS, a `dict` is its own beast, and is not the same as an object obtained by instantiating a Python class.

Creation:

```
1 o1 = {}          # empty dict
2 o2 = {"x": 12}    # one key
3 o3 = {y: "hello"} # ERROR: key quotes required,
4                  # unless y is a variable
5                  # that holds a value you wish
6                  # to use as a key
7
8 o4 = {  # multiline format
9     "a": 20,
10    "b": 1.2,
11    "foo": "hello"
12 }
```

Access:

```
print(o4["a"])  # Prints 20
```

Dot notation does not work with Python dicts.

String Formatting

JavaScript

Converting to different number bases:

```
1 let x = 237;
2 let x_binary = x.toString(2); // string '11101101'
3 let x_hex = x.toString(16);   // string 'ed'
```

Controlling floating point precision:

```
1 let x = 3.1415926535;
2 let y = x.toFixed(2); // string '3.14'
```

Padding and justification:

```
1 let s = "Hello!";
2 let t = s.padStart(10, ' '); // string '    Hello!'
3 let u = s.padEnd(10, ' '); // string 'Hello!    '
4
5 let v = s.padStart(10, '*'); // string '****Hello!'
6
7 // Pad with leading zeroes
8 (12).toString(2).padStart(8, '0'); // string '00001100'
```

Parameterized strings:

```
1 let x = 3.1415926;
2 let y = "Hello";
3 let z = 67;
4
5 // 'x is 3.14, y is "Hello", z is 01000011'
6 let s = `x is ${x.toFixed(2)}, y is "${y}", z is ${z.toString(2).padStart(8, '0')}`
```

Python

Python has the printf operator `%` which is tremendously powerful. (If the operands to `%` are numbers, modulo is performed. If the left operand is a string, printf is performed.)

But even `%` is being superseded by the `format` function.

[Tons of details at pyformat.info.](#)

Also see [printf-style String Formatting](#) for a reference.

Booleans and Conditionals

JavaScript

Literal boolean values:

```
1 x = true;
2 y = false;
```

Boolean operators:

Operator	Definition
<code>==</code>	Equality
<code>!=</code>	Inequality
<code>===</code>	Strict equality
<code>!==</code>	Strict inequality
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal

The concept of strict equality/inequality applies to items that might normally be converted into a compatible type. The strict tests will consider if the types themselves are the same.

```
1 0 == "0"; // true
2 0 === "0"; // false
3
4 0 == []; // true
5 0 === []; // false
6
```

```
7 0 == 0; // true
8 0 === 0; // true
```

Logical operators:

Operator	Description
!	Logical inverse, not
&&	Logical AND
&	Logical AND

The not operator `!` can be used to test whether or not a value is "truthy".

```
1 !0; // true
2 !!0; // false
3 !!; // false
4 !null; // true
5 !"0"; // false, perhaps unexpectedly
6 !"x"; // false
```

Example:

```
1 if (a == 2 && b !== "") {
2   // Something complicated
3 }
```

Python

Literal boolean values:

```
1 x = True
2 y = False
```

Boolean operators:

Operator	Definition
<code>==</code>	Equality
<code>!=</code>	Inequality
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal

Logical operators:

Operator	Description
not	Logical inverse, not
and	Logical AND
or	Logical OR

The `not` operator can be used to test whether or not a value is "truthy".

```

1 not 0      # true
2 not not 0  # false
3 not 1      # false
4 not None;  # true
5 not "0"    # false, perhaps unexpectedly
6 not "x"    # false

```

Example:

```

1 if a == 2 and b != "":
2   # Something complicated

```

for Loops

JavaScript

C-style `for` loops:

```

1 for (let i = 0; i < 10; i++) {
2   console.log(i);
3 }

```

`for - in` loops iterate over the properties of an object or indexes of an array:

```

1 a = [10, 20, 30];
2
3 for (let i in a) {
4   console.log(i);    # 0 1 2
5   console.log(a[i]); # 10 20 30
6 }
7
8 b = {'x': 77, 'y': 88, 'z': 99};
9
10 for (let i in b) {
11   console.log(i);    # x y z
12   console.log(b[i]); # 77 88 99
13 }

```

`for - of` loops access the values within the array (as opposed to the indexes of the array):

```

1 a = [10, 20, 30];
2

```

```
3  for (let i of a) {  
4    console.log(i); # 10 20 30  
5  }
```

Python

`for - in` loops over an *iteratable*. This can be a list, object, or other type of iterable item.

Counting loops:

```
1 # Use the range() function to count:  
2 for i in range(10):  
3   print(i) # Prints 0-9  
4  
5 for i in range(20, 30):  
6   print(i) # Prints 20-29  
7  
8 for i in range(-10, 20, 3):  
9   print(i) # Print every 3rd number from -10 to 19
```

Iterating over other types:

```
1 # A list  
2 a = [10, 20, 30]  
3  
4 # Print 10 20 30  
5 for i in a:  
6   print(i)  
7  
8 # A dict  
9 b = {'x':5, 'y':15, 'z':0}  
10  
11 # Print x y z (the keys of the dict)  
12 for i in b:  
13   print(i)
```

while Loops

JavaScript

C-style `while` and `do - while`:

```
1 // Print 10 down to 0:  
2  
3 let x = 10;  
4 while (x >= 0) {  
5   console.log(x);  
6   x--;  
7 }  
8  
9 // Print 0 up to 9:  
10 let x = 0;  
11 do {  
12   console.log(x);  
13   x++;  
14 } while (x < 10);
```

Python

Python has a `while` loop:

```
1 # Print from 10 down through 0
2 x = 10
3 while x >= 0:
4     print(x)
5     x -= 1
```

switch Statement

JavaScript

JS can switch on various data types:

```
1 switch(x) {
2     case "foo":
3         console.log("x is foo, all right");
4         break;
5     case 23:
6         console.log("but here x is 23");
7         break;
8     default:
9         console.log("x is something else entirely");
10 }
```

Python

Python doesn't have a `switch` statement. You can use `if - elif - else` blocks.

A somewhat clumsy approximation of a `switch` can be constructed with a `dict` of functions.

```
1 def func1():
2     print("case 1 is hit")
3
4 def func2():
5     print("case 2 is hit")
6
7 def func3():
8     print("case 3 is hit")
9
10 funcs = {
11     "alpha": func1,
12     "bravo": func2,
13     "charlie": func3
14 };
15
16 x = "bravo"
17 funcs[x]() # calls func2
```

if Conditionals

JavaScript

JS uses C-style `if` statements:

```
1 if (x == 10) {
```

```
2   console.log("x is 10");
3 } else if (x == 20) {
4   console.log("x is 20");
5 } else {
6   console.log("x is something else");
7 }
```

Python

Python notably uses `elif` instead of `else if`.

```
1 if x == 10:
2   print("x is 10")
3 elif x == 20:
4   print("x is 20")
5 else:
6   print("x is something else")
```

Classes

JavaScript

The current object is referred to by `this`.

Pre ES-2015, classes were created using functions. This is now outdated.

```
1 function Goat(color) {
2   this.legs = 4;
3   this.color = color;
4 }
5
6 g = new Goat("brown");
```

JS uses prototypal inheritance. Pre ES-2015, this was explicit, and is also outdated:

```
1 function Creature(type) {
2   this.type = type;
3 }
4
5 // Make Goats inherit from Creature:
6 Goat.prototype = new Creature("mammal");
7
8 // Add a method:
9 Goat.prototype.jump = function () {
10   console.log("I'm jumping! Yay!");
11 };
12
13 g = new Goat("red");
14 g.type; // "mammal", since Goat inherits from Creature
15 g.jump(); // "I'm jumping! Yay!"
```

Modern JS introduced the `class` keyword and a syntax more familiar to most other OOP languages. Note that the inheritance model is still prototypal inheritance; it's just that the details are hidden from the developer.

```
1 class Creature {
2   constructor(type) {
```

```

3     this.type = type;
4   }
5 }
6
7 class Goat extends Creature {
8   constructor(color) {
9     super("mammal");
10    this.legs = 4;
11    this.color = color;
12  }
13
14 jump() {
15   console.log("I'm jumping! Yay!");
16 }
17 }
18
19 g = new Goat("orange");
20 g.type; // "mammal"
21 g.jump(); // "I'm jumping! Yay!"

```

JS does not support multiple inheritance since each object can only have one prototype object. You have to use a *mix-in* if you want to achieve similar functionality.

Python

The current object is referred to by `self`. Note that `self` is explicitly present as the first parameter in object methods.

Python 2 syntax:

```

1 class Creature:
2   def __init__(self, type): # constructor
3     self.type = type
4
5 class Goat(Creature):
6   def __init__(self, color):
7     # call super constructor
8     Creature.__init__(self, "mammal")
9     self.color = color
10
11  def jump(self):
12    print("I'm jumping! Yay!")
13
14 g = Goat("green")
15 g.type # mammal
16 g.jump() # I'm jumping! Yay!

```

Python 3 syntax includes the new `super()` keyword to make life easier.

```

1 class Creature:
2   def __init__(self, type): # constructor
3     self.type = type
4
5 class Goat(Creature):
6   def __init__(self, color):
7     # call super constructor
8     super().__init__("mammal") # <-- Nicer!
9     self.color = color
10
11  def jump(self):
12    print("I'm jumping! Yay!")
13
14 g = Goat("green")
15 g.type # mammal
16 g.jump() # I'm jumping! Yay!

```

Python supports multiple inheritance.

Misc. Resources

 [Python Cheat Sheet](#)

00_Python_Cheatsheet.ipynb - 157KB

 [GitHub: Web-Dev-Collaborative/jupyter-learn-py/aa9ad2b2ffa97ac278c13ebca6b89e4c1d1aad08](#)

<https://mybinder.org/v2/gh/Web-Dev-Collaborative/jupyter-learn-py/aa9ad2b2ffa97ac278c13ebca6b89e4c1d1aad08>

Intro To Python w Jupyter Notebooks



intro-2-python-w-jupyter

<https://gist.github.com/bgoonz/4f5c0b5fe80a84421ff9a5a66dc>
e29da

Calculating Big O

Computing Big O

Brian "Beej Jorgensen" Hall edited this page on Nov 4, 2019 · 9 revisions

Goal: determine how runtime/number of operations scales up as the input scales up. How much longer does it take to run as the size of the data to process gets bigger?

Steps to compute Big O

- Things in sequence that *aren't* loops add together
 - A single thing inside a loop gets multiplied by the loop
1. Go a line at a time, only looking at lines that are executable
 2. Add all the things in sequence that you can first
 3. Then multiply by the loops
 4. Then repeat steps 2-3 as many times as you can
 5. Then keep the dominant term from the resulting sum(s)
 6. Then drop constants

Hints

- If you have something that's `O(number_of_elements_in_the_data)`, we use `n` as shorthand for `number_of_elements_in_the_data`, so `O(n)`.
- Individual statements tend to be `O(1)`.
- Loop statements tend to be `O(how-many-times-they-loop)`.
- Anything that doubles the runtime each step is `O(2^n)` (e.g. naive Fibonacci).
- Anything that triples the runtime each step is `O(3^n)`.
- Anything that halves the runtime each step is `O(log n)` (e.g. binary search).
- By *dominant term* we mean, "thing which is largest given some large value of n , like 10000". `O(n)` dominates `O(1)`. `O(n^2)` dominates `O(n)` and `O(1)`.
- Loops that iterate over entire lists are `O(n)`, where `n` is the size of the list.
- But loops that binary search over a list are `O(log n)`!

Recursion

- Recursive functions are like loops, where the body of the function is the body of the loop.
- You need to figure out how many times the function will call itself, and that's the Big O that you need to multiply against the Big O of the function body.
- Keep in mind that recursion comes with an inherent memory cost that loops don't incur, since each recursive call adds an additional execution frame to the stack; in other words, calling a function is not free!

Gotchas

- Built in functions might incur significant Big O without you noticing. Python's list `.copy()` might seem like just a simple `O(1)` line, but it's `O(n)` under the hood.
- Beware of loops that modify their index in weird ways.

Example

Label all statements by their time complexities. Individual statements get their Big O, while loops get the number of times they loop.

```

1 def foo(n):
2     a = 10                 # O(1)
3     b = 20                 # O(1)
4
5     for i in range(n):    # O(n)
6         a += b             # O(1)
7         b += 1              # O(1)
8
9     for j in range(n**2): # O(n^2)
10        a -= 2            # O(1)
11        print(a)          # O(1)
12
13        for k in range(n/2): # O(n/2)
14            print(k)          # O(1)
15
16    return a + b           # O(1)

```

Now we'll replace the lines of code with their Big O for brevity:

```

1 def foo(n):
2     # O(1)
3     # O(1)
4
5     # O(n)
6     # O(1)
7     # O(1)
8
9     # O(n^2)
10    # O(1)
11    # O(1)
12
13    # O(n/2)
14    # O(1)
15
16    # O(1)

```

Try to add things in sequence, but remember that loops interrupt sequences!

```

1 def foo(n):
2     # O(2)      -- was O(1) + O(1)
3
4     # O(n)
5     # O(2)      -- was O(1) + O(1)
6
7     # O(n^2)
8     # O(2)      -- was O(1) + O(1)
9
10    # O(n/2)
11    # O(1)
12
13    # O(1)

```

Now we see if we can multiply any loops by their bodies.

```

1 def foo(n):
2     # O(2)
3
4     # O(2 * n)   -- had body O(2)
5
6     # O(n^2)      -- can't do this one yet--body has more than one item
7     # O(2)
8

```

```
9         # O(1 * n/2)    -- had body O(1)
10
11     # O(1)
```

Let's try to add any sequences again.

```
1 def foo(n):
2     # O(2 + 2 * n)
3
4     # O(n^2)
5     # O(2 + 1 * n/2)
6
7     # O(1)
```

Now let's try multiplying loops again

```
1 def foo(n):
2     # O(2 + 2 * n)
3
4     # O(n^2 * (2 + 1 * n/2))
5
6     # O(1)
```

Add add sequences again:

```
1 def foo(n):
2     # O((2 + 2 * n) + (n^2 * (2 + 1 * n/2)) + 1)
```

Now we're down to one line. That's the time complexity, but we need to reduce it.

Break out your algebra.

```
1 (2 + 2 * n) + (n^2 * (2 + 1 * n/2)) + 1   From the Big O, above
2 2 + 2 * n + n^2 * (2 + 1 * n/2) + 1   Lose unnecessary parens
3 3 + 2 * n + n^2 * (2 + 1 * n/2)   Add some like terms
4 3 + 2 * n + n^2 * (2 + n/2)   1* is does nothing
5 3 + 2 * n + 2 * n^2 + n/2 * n^2   Distribute n^2
6 3 + 2 * n + 2 * n^2 + 1/2 * n * n^2   Note the n/2 is 1/2*n
7 3 + 2 * n + 2 * n^2 + 1/2 * n^3   n * n^2 = n^3
8 (3) + (2 * n) + (2 * n^2) + (1/2 * n^3)   Choose the most dominant from the sum
9 1/2 * n^3   1/2 * n^3 grows fastest, is dominant
10 n^3   Drop the constant
```

$O(n^3)$ is the time complexity of this function.

With practice, you can do this in your head. Looking back, the nested loop *must* have been where the function spent the most of its time; an experienced dev would see that and just quickly compute the Big O for that function from that nested loop alone.

Example with two variables

```
1 def foo(m, n):
2     for i in range(m * n):
```

```
3     print(i)
```

When you have two inputs like this, there's no way to reduce it farther than $O(m \times n)$ (or $O(n \times m)$, same thing). That's the answer.

Sometimes when m and n tend to be roughly similar, developers will casually say this is $O(n^2)$, but it's really $O(m \times n)$.

Example with lists

```
1 def foo(x): # x is a list
2     for i in x: # O(n)
3         print(i) # O(1)
```

In this example, we're not explicitly passing in an n parameter, but rather a list.

The list has a number of elements in it, and we refer to this number as n by default.

The `for` loop is therefore $O(n)$, because it will iterate one time for each element in the list.

Another example:

```
1 def foo(x, y): # x and y are lists
2     for i in x: # O(n)
3         for j in y: # O(m)
4             print(i, j) # O(1)
```

Here we've used n to represent the number of elements in list x , and m to represent the number in list y .

We can use our simplification rules and see that the entire function is $O(n \times m \times 1)$, or $O(n \times m)$. (Or $O(n^2)$ if we're speaking informally, and assuming that n and m are very similar.)

Example with trivial recursion

```
1 def foo(x): # x is a list
2     if len(x) == 0:
3         return
4
5     print(x[0])
6     foo(x[1:])
```

The above function prints out every element in a list. But it's trickier to see what the Big O is. Our normal rules don't work entirely well.

The secret is that recursive functions are like loops on steroids. So you know it's similar to a loop in that it's going to perform a number of operations. But how many? n ? n^2 ? We have to figure it out.

In the above example, each call to `foo()` results in *one* more call to `foo()`. (Because we look in the body of the function and we see it only calls itself once.) And it's going to keep calling itself a number of times. *How many times will `foo()` call itself?*

Here, if we declare that n is the number of elements in list x , `foo()` calls itself n times, once for each element in x .

So the recursion itself, acting like a loop, is $O(n)$.

We still have to look at the things *inside* `foo()` to see what else is going on. The body of `foo()` becomes like the body of the loop.

But it looks like in there we only have a couple `O(1)` things, so the whole thing becomes `O(n * (1 + 1))`, aka `O(2n)` AKA `O(n)`. Final answer.

Example with Fibonacci

```
1 def fib(n):    # Give me the nth Fibonacci number
2     if n < 2:
3         return n
4
5     return fib(n-2) + fib(n-1)  # Calls itself twice per call!
```

Again, think loop on steroids. `fib()` calls itself... but it calls itself *two* times per call. ...*ish*.

We call it `1` time, it calls itself `2` times. Those `2` times call it `4` times, which call it `8` times, which call it `16` times, etc. If you recognize those numbers, you'll know those are powers of 2. $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, and all the way up to 2^n .

This is an `O(2^n)` recursive call. (With an `O(1)` body.)

Sure, `fib(n-2)` only calls it $1/2 \times n$ times, but we chuck that constant for Big O.

And the base case won't necessarily let `n` get all the way down to zero, but those are just some `-1`s or `-2`s, and those terms aren't dominant in Big O.

Python Cheat Sheet



<https://websitesetup.org/wp-content/uploads/2021/04/Python-cheat-sheet-April-2021.pdf>

<https://websitesetup.org/wp-content/uploads/2021/04/Python-cheat-sheet-April-2021.pdf>



Beginner's Python Cheat Sheet

Variables and Strings
Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

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

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists
A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples
Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements
If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals      x == 42
not equal   x != 42
greater than x > 42
or equal to x >= 42
less than   x < 42
or equal to x <= 42
```

Conditional test with lists

```
'trek' in bikes
'sunly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries
Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite!')
```

User input
Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)
```

pi = input("What's the value of pi? ")
pi = float(pi)

Python Crash Course 
Covers Python 3 and Python 2
nostarchpress.com/pythoncrashcourse



[beginners_python_cheat_sheet_pcc_all.pdf](#)

beginners_python_cheat_sheet_pcc_all.pdf - 2MB



GitHub: [wilfredinni/python-cheatsheet](https://github.com/wilfredinni/python-cheatsheet)/1b1fd1d46ea6b2bff715db5c0e1c2b26bacb74f5

<https://mybinder.org/v2/gh/wilfredinni/python-cheatsheet/1b1fd1d46ea6b2bff715db5c0e1c2b26bacb74f5>

Python Basics

Math Operators

From **Highest** to **Lowest** precedence:

Operators	Operation	Example
**	Exponent	<code>2 ** 3 = 8</code>
%	Modulus/Remainder	<code>22 % 8 = 6</code>
//	Integer division	<code>22 // 8 = 2</code>
/	Division	<code>22 / 8 = 2.75</code>
*	Multiplication	<code>3 * 3 = 9</code>
-	Subtraction	<code>5 - 2 = 3</code>
+	Addition	<code>2 + 2 = 4</code>

Examples of expressions in the interactive shell:

```
1 >>> 2 + 3 * 6
2 20
```

```
1 >>> (2 + 3) * 6
2 30
```

```
1 >>> 2 ** 8
2 256
```

```
1 >>> 23 // 7
2 3
```

```
1 >>> 23 % 7
2 2
```

```
1 >>> (5 - 1) * ((7 + 1) / (3 - 1))
2 16.0
```

Data Types

Data Type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

String Concatenation and Replication

String concatenation:

```
1 >>> 'Alice' 'Bob'
2 'AliceBob'
```

Note: Avoid `+` operator for string concatenation. Prefer string formatting.

String Replication:

```
1 >>> 'Alice' * 5
2 'AliceAliceAliceAliceAlice'
```

Variables

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (`_`) character.
3. It can't begin with a number.
4. Variable name starting with an underscore (`_`) are considered as "unuseful".

Example:

```
1 >>> spam = 'Hello'
2 >>> spam
3 'Hello'
```

```
>>> _spam = 'Hello'
```

`_spam` should not be used again in the code.

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

```
1 # This is a
2 # multiline comment
```

Code with a comment:

```
a = 1 # initialization
```

Please note the two spaces in front of the comment.

Function docstring:

```
1 def foo():
2     """
3     This is a function docstring
4     You can also use:
5     ''' Function Docstring '''
6     """
```

The `print()` Function

```
1 >>> print('Hello world!')
2 Hello world!
```

```
1 >>> a = 1
2 >>> print('Hello world!', a)
3 Hello world! 1
```

The `input()` Function

Example Code:

```
1 >>> print('What is your name?')    # ask for their name
2 >>> myName = input()
3 >>> print('It is good to meet you, {}'.format(myName))
4 What is your name?
5 Al
6 It is good to meet you, Al
```

The `len()` Function

Evaluates to the integer value of the number of characters in a string:

```
1 >>> len('hello')
2 5
```

Note: test of emptiness of strings, lists, dictionary, etc, should **not** use len, but prefer direct boolean evaluation.

```
1 >>> a = [1, 2, 3]
2 >>> if a:
3 >>>     print("the list is not empty!")
```

The str(), int(), and float() Functions

Integer to String or Float:

```
1 >>> str(29)
2 '29'
```

```
1 >>> print('I am {} years old.'.format(str(29)))
2 I am 29 years old.
```

```
1 >>> str(-3.14)
2 '-3.14'
```

Float to Integer:

```
1 >>> int(7.7)
2 7
```

```
1 >>> int(7.7) + 1
2 8
```

Flow Control

Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater Than
<code><=</code>	Less than or Equal to
<code>>=</code>	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
1 >>> 42 == 42
2 True
```

```
1 >>> 40 == 42
2 False
```

```
1 >>> 'hello' == 'hello'
2 True
```

```
1 >>> 'hello' == 'Hello'
2 False
```

```
1 >>> 'dog' != 'cat'
2 True
```

```
1 >>> 42 == 42.0
2 True
```

```
1 >>> 42 == '42'
2 False
```

Boolean evaluation

Never use `==` or `!=` operator to evaluate boolean operation. Use the `is` or `is not` operators, or use implicit boolean evaluation.

NO (even if they are valid Python):

```
1 >>> True == True
2 True
```

```
1 >>> True != False
2 True
```

YES (even if they are valid Python):

```
1 >>> True is True
2 True
```

```
1 >>> True is not False
2 True
```

These statements are equivalent:

```
1 >>> if a is True:
2 >>>     pass
3 >>> if a is not False:
4 >>>     pass
5 >>> if a:
6 >>>     pass
```

And these as well:

```
1 >>> if a is False:
2 >>>     pass
3 >>> if a is not True:
4 >>>     pass
5 >>> if not a:
6 >>>     pass
```

Boolean Operators

There are three Boolean operators: and, or, and not.

The *and* Operator's *Truth Table*:

Expression	Evaluates to
True and True	True
True and False	False
False and True	False
False and False	False

The *or* Operator's *Truth Table*:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The *not* Operator's *Truth Table*:

Expression	Evaluates to
not True	False

```
not False
```

```
True
```

Mixing Boolean and Comparison Operators

```
1 >>> (4 < 5) and (5 < 6)
2 True
```

```
1 >>> (4 < 5) and (9 < 6)
2 False
```

```
1 >>> (1 == 2) or (2 == 2)
2 True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
1 >>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
2 True
```

if Statements

```
1 if name == 'Alice':
2     print('Hi, Alice.')
```

else Statements

```
1 name = 'Bob'
2 if name == 'Alice':
3     print('Hi, Alice.')
4 else:
5     print('Hello, stranger.')
```

elif Statements

```
1 name = 'Bob'
2 age = 5
3 if name == 'Alice':
4     print('Hi, Alice.')
5 elif age < 12:
6     print('You are not Alice, kiddo.')
```

```
1 name = 'Bob'
2 age = 30
3 if name == 'Alice':
4     print('Hi, Alice.')
5 elif age < 12:
6     print('You are not Alice, kiddo.')
```

```
7 else:  
8     print('You are neither Alice nor a little kid.')
```

while Loop Statements

```
1 spam = 0  
2 while spam < 5:  
3     print('Hello, world.')  
4     spam = spam + 1
```

break Statements

If the execution reaches a break statement, it immediately exits the while loop's clause:

```
1 while True:  
2     print('Please type your name.')  
3     name = input()  
4     if name == 'your name':  
5         break  
6 print('Thank you!')
```

continue Statements

When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop.

```
1 while True:  
2     print('Who are you?')  
3     name = input()  
4     if name != 'Joe':  
5         continue  
6     print('Hello, Joe. What is the password? (It is a fish.)')  
7     password = input()  
8     if password == 'swordfish':  
9         break  
10    print('Access granted.')
```

for Loops and the range() Function

```
1 >>> print('My name is')  
2 >>> for i in range(5):  
3 >>>     print('Jimmy Five Times {}'.format(str(i)))  
4 My name is  
5 Jimmy Five Times (0)  
6 Jimmy Five Times (1)  
7 Jimmy Five Times (2)  
8 Jimmy Five Times (3)  
9 Jimmy Five Times (4)
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

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

```
4 2
5 4
6 6
7 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
1 >>> for i in range(5, -1, -1):
2 >>>     print(i)
3 5
4 4
5 3
6 2
7 1
8 0
```

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
1 >>> for i in [1, 2, 3, 4, 5]:
2 >>>     if i == 3:
3 >>>         break
4 >>> else:
5 >>>     print("only executed when no item of the list is equal to 3")
```

Importing Modules

```
1 import random
2 for i in range(5):
3     print(random.randint(1, 10))
```

```
import random, sys, os, math
```

```
from random import *
```

Ending a Program Early with `sys.exit()`

```
1 import sys
2
3 while True:
4     print('Type exit to exit.')
5     response = input()
6     if response == 'exit':
7         sys.exit()
8     print('You typed {}'.format(response))
```

Functions

```
1 >>> def hello(name):
2 >>>     print('Hello {}'.format(name))
3 >>>
4 >>> hello('Alice')
5 >>> hello('Bob')
6 Hello Alice
7 Hello Bob
```

Return Values and return Statements

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

- The `return` keyword.
- The value or expression that the function should return.

```
1 import random
2 def getAnswer(answerNumber):
3     if answerNumber == 1:
4         return 'It is certain'
5     elif answerNumber == 2:
6         return 'It is decidedly so'
7     elif answerNumber == 3:
8         return 'Yes'
9     elif answerNumber == 4:
10        return 'Reply hazy try again'
11    elif answerNumber == 5:
12        return 'Ask again later'
13    elif answerNumber == 6:
14        return 'Concentrate and ask again'
15    elif answerNumber == 7:
16        return 'My reply is no'
17    elif answerNumber == 8:
18        return 'Outlook not so good'
19    elif answerNumber == 9:
20        return 'Very doubtful'
21
22 r = random.randint(1, 9)
23 fortune = getAnswer(r)
24 print(fortune)
```

The None Value

```
1 >>> spam = print('Hello!')
2 Hello!
```

```
1 >>> spam is None
2 True
```

Note: never compare to `None` with the `==` operator. Always use `is`.

Keyword Arguments and `print()`

```
1 >>> print('Hello', end='')
2 >>> print('World')
3 HelloWorld
```

```
1 >>> print('cats', 'dogs', 'mice')
2 cats dogs mice
```

```
1 >>> print('cats', 'dogs', 'mice', sep=',')
2 cats,dogs,mice
```

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

The global Statement

If you need to modify a global variable from within a function, use the global statement:

```
1 >>> def spam():
2 >>>     global eggs
3 >>>     eggs = 'spam'
4 >>>
5 >>>     eggs = 'global'
6 >>>     spam()
7 >>>     print(eggs)
8 spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Exception Handling

Basic exception handling

```
1 >>> def spam(divideBy):
2 >>>     try:
3 >>>         return 42 / divideBy
4 >>>     except ZeroDivisionError as e:
5 >>>         print('Error: Invalid argument: {}'.format(e))
6 >>>
7 >>> print(spam(2))
8 >>> print(spam(12))
9 >>> print(spam(0))
10 >>> print(spam(1))
11 21.0
12 3.5
13 Error: Invalid argument: division by zero
14 None
15 42.0
```

Final code in exception handling

Code inside the `finally` section is always executed, no matter if an exception has been raised or not, and even if an exception is not caught.

```
1 >>> def spam(divideBy):
2   >>>     try:
3   >>>         return 42 / divideBy
4   >>>     except ZeroDivisionError as e:
5   >>>         print('Error: Invalid argument: {}'.format(e))
6   >>>     finally:
7   >>>         print("-- division finished --")
8   >>> print(spam(2))
9 -- division finished --
10 21.0
11 >>> print(spam(12))
12 -- division finished --
13 3.5
14 >>> print(spam(0))
15 Error: Invalid Argument division by zero
16 -- division finished --
17 None
18 >>> print(spam(1))
19 -- division finished --
20 42.0
```

Lists

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2
3 >>> spam
4 ['cat', 'bat', 'rat', 'elephant']
```

Getting Individual Values in a List with Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[0]
3 'cat'
```

```
1 >>> spam[1]
2 'bat'
```

```
1 >>> spam[2]
2 'rat'
```

```
1 >>> spam[3]
2 'elephant'
```

Negative Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
2 >>> spam[-1]
3 'elephant'
```

```
1 >>> spam[-3]
2 'bat'
```

```
1 >>> 'The {} is afraid of the {}.'.format(spam[-1], spam[-3])
2 'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[0:4]
3 ['cat', 'bat', 'rat', 'elephant']
```

```
1 >>> spam[1:3]
2 ['bat', 'rat']
```

```
1 >>> spam[0:-1]
2 ['cat', 'bat', 'rat']
```

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[:2]
3 ['cat', 'bat']
```

```
1 >>> spam[1:]
2 ['bat', 'rat', 'elephant']
```

Slicing the complete list will perform a copy:

```
1 >>> spam2 = spam[:]
2 ['cat', 'bat', 'rat', 'elephant']
3 >>> spam.append('dog')
4 >>> spam
5 ['cat', 'bat', 'rat', 'elephant', 'dog']
6 >>> spam2
7 ['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with `len()`

```
1 >>> spam = ['cat', 'dog', 'moose']
2 >>> len(spam)
3 3
```

Changing Values in a List with Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[1] = 'aardvark'
3
4 >>> spam
5 ['cat', 'aardvark', 'rat', 'elephant']
6
7 >>> spam[2] = spam[1]
8
9 >>> spam
10 ['cat', 'aardvark', 'aardvark', 'elephant']
11
12 >>> spam[-1] = 12345
13
14 >>> spam
15 ['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

```
1 >>> [1, 2, 3] + ['A', 'B', 'C']
2 [1, 2, 3, 'A', 'B', 'C']
3
4 >>> ['X', 'Y', 'Z'] * 3
5 ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
6
7 >>> spam = [1, 2, 3]
8
9 >>> spam = spam + ['A', 'B', 'C']
10
11 >>> spam
12 [1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> del spam[2]
3 >>> spam
4 ['cat', 'bat', 'elephant']
```

```
1 >>> del spam[2]
2 >>> spam
3 ['cat', 'bat']
```

Using for Loops with Lists

```
1 >>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
2 >>> for i, supply in enumerate(supplies):
3 >>>     print('Index {} in supplies is: {}'.format(str(i), supply))
4 Index 0 in supplies is: pens
5 Index 1 in supplies is: staplers
6 Index 2 in supplies is: flame-throwers
7 Index 3 in supplies is: binders
```

Looping Through Multiple Lists with zip()

```
1 >>> name = ['Pete', 'John', 'Elizabeth']
2 >>> age = [6, 23, 44]
3 >>> for n, a in zip(name, age):
4     print('{} is {} years old'.format(n, a))
5 Pete is 6 years old
6 John is 23 years old
7 Elizabeth is 44 years old
```

The in and not in Operators

```
1 >>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
2 True
```

```
1 >>> spam = ['hello', 'hi', 'howdy', 'heyas']
2 >>> 'cat' in spam
3 False
```

```
1 >>> 'howdy' not in spam
2 False
```

```
1 >>> 'cat' not in spam
2 True
```

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
1 >>> cat = ['fat', 'orange', 'loud']
2
3 >>> size = cat[0]
4
5 >>> color = cat[1]
6
7 >>> disposition = cat[2]
```

You could type this line of code:

```
1 >>> cat = ['fat', 'orange', 'loud']
2
3 >>> size, color, disposition = cat
```

The multiple assignment trick can also be used to swap the values in two variables:

```
1 >>> a, b = 'Alice', 'Bob'
2 >>> a, b = b, a
3 >>> print(a)
4 'Bob'
```

```
1 >>> print(b)
2 'Alice'
```

Augmented Assignment Operators

Operator	Equivalent
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1
spam %= 1	spam = spam % 1

Examples:

```
1 >>> spam = 'Hello'
2 >>> spam += ' world!'
3 >>> spam
4 'Hello world!'
5
6 >>> bacon = ['Zophie']
7 >>> bacon *= 3
8 >>> bacon
9 ['Zophie', 'Zophie', 'Zophie']
```

Finding a Value in a List with the index() Method

```
1 >>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
2
3 >>> spam.index('Pooka')
4 1
```

Adding Values to Lists with the append() and insert() Methods

append():

```
1 >>> spam = ['cat', 'dog', 'bat']
2
3 >>> spam.append('moose')
4
5 >>> spam
6 ['cat', 'dog', 'bat', 'moose']
```

insert():

```
1 >>> spam = ['cat', 'dog', 'bat']
2
```

```
3 >>> spam.insert(1, 'chicken')
4
5 >>> spam
6 ['cat', 'chicken', 'dog', 'bat']
```

Removing Values from Lists with remove()

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2
3 >>> spam.remove('bat')
4
5 >>> spam
6 ['cat', 'rat', 'elephant']
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

Sorting the Values in a List with the sort() Method

```
1 >>> spam = [2, 5, 3.14, 1, -7]
2 >>> spam.sort()
3 >>> spam
4 [-7, 1, 2, 3.14, 5]
```

```
1 >>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
2 >>> spam.sort()
3 >>> spam
4 ['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order:

```
1 >>> spam.sort(reverse=True)
2 >>> spam
3 ['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

If you need to sort the values in regular alphabetical order, pass str.lower for the key keyword argument in the sort() method call:

```
1 >>> spam = ['a', 'z', 'A', 'Z']
2 >>> spam.sort(key=str.lower)
3 >>> spam
4 ['a', 'A', 'z', 'Z']
```

You can use the built-in function `sorted` to return a new list:

```
1 >>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
2 >>> sorted(spam)
3 ['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

Tuple Data Type

```
1 >>> eggs = ('hello', 42, 0.5)
2 >>> eggs[0]
3 'hello'
```

```
1 >>> eggs[1:3]
2 (42, 0.5)
```

```
1 >>> len(eggs)
2 3
```

The main way that tuples are different from lists is that tuples, like strings, are immutable.

Converting Types with the `list()` and `tuple()` Functions

```
1 >>> tuple(['cat', 'dog', 5])
2 ('cat', 'dog', 5)
```

```
1 >>> list(('cat', 'dog', 5))
2 ['cat', 'dog', 5]
```

```
1 >>> list('hello')
2 ['h', 'e', 'l', 'l', 'o']
```

Dictionaries and Structuring Data

Example Dictionary:

```
myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

The `keys()`, `values()`, and `items()` Methods

`values()`:

```
1 >>> spam = {'color': 'red', 'age': 42}
2 >>> for v in spam.values():
3     print(v)
4 red
5 42
```

`keys()`:

```
1 >>> for k in spam.keys():
2     print(k)
```

```
3 color
4 age
```

items():

```
1 >>> for i in spam.items():
2 >>>     print(i)
3 ('color', 'red')
4 ('age', 42)
```

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.

```
1
2 >>> spam = {'color': 'red', 'age': 42}
3 >>>
4 >>> for k, v in spam.items():
5 >>>     print('Key: {} Value: {}'.format(k, str(v)))
6 Key: age Value: 42
7 Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
1 >>> 'name' in spam.keys()
2 True
```

```
1 >>> 'Zophie' in spam.values()
2 True
```

```
1 >>> # You can omit the call to keys() when checking for a key
2 >>> 'color' in spam
3 False
```

```
1 >>> 'color' not in spam
2 True
```

The get() Method

Get has two parameters: key and default value if the key did not exist

```
1 >>> picnic_items = {'apples': 5, 'cups': 2}
2
3 >>> 'I am bringing {} cups.'.format(str(picnic_items.get('cups', 0)))
4 'I am bringing 2 cups.'
```

```
1 >>> 'I am bringing {} eggs.'.format(str(picnic_items.get('eggs', 0)))
2 'I am bringing 0 eggs.'
```

The `setdefault()` Method

Let's consider this code:

```
1 spam = {'name': 'Pooka', 'age': 5}
2
3 if 'color' not in spam:
4     spam['color'] = 'black'
```

Using `setdefault` we could write the same code more succinctly:

```
1 >>> spam = {'name': 'Pooka', 'age': 5}
2 >>> spam.setdefault('color', 'black')
3 'black'
```

```
1 >>> spam
2 {'color': 'black', 'age': 5, 'name': 'Pooka'}
```

```
1 >>> spam.setdefault('color', 'white')
2 'black'
```

```
1 >>> spam
2 {'color': 'black', 'age': 5, 'name': 'Pooka'}
```

Pretty Printing

```
1 >>> import pprint
2 >>>
3 >>> message = 'It was a bright cold day in April, and the clocks were striking
4 >>> thirteen.'
5 >>> count = {}
6 >>>
7 >>> for character in message:
8 >>>     count.setdefault(character, 0)
9 >>>     count[character] = count[character] + 1
10 >>>
11 >>> pprint.pprint(count)
12 {' ': 13,
13 ',': 1,
14 '.': 1,
15 'A': 1,
16 'I': 1,
17 'a': 4,
18 'b': 1,
19 'c': 3,
20 'd': 3,
21 'e': 5,
22 'g': 2,
23 'h': 3,
```

```
24 'i': 6,
25 'k': 2,
26 'l': 3,
27 'n': 4,
28 'o': 2,
29 'p': 1,
30 'r': 5,
31 's': 3,
32 't': 6,
33 'w': 2,
34 'y': 1}
```

Merge two dictionaries

```
1 # in Python 3.5+:
2 >>> x = {'a': 1, 'b': 2}
3 >>> y = {'b': 3, 'c': 4}
4 >>> z = {**x, **y}
5 >>> z
6 {'c': 4, 'a': 1, 'b': 3}
7
8 # in Python 2.7
9 >>> z = dict(x, **y)
10 >>> z
11 {'c': 4, 'a': 1, 'b': 3}
```

sets

From the Python 3 documentation

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.

Initializing a set

There are two ways to create sets: using curly braces `{}` and the built-in function `set()`

```
1 >>> s = {1, 2, 3}
2 >>> s = set([1, 2, 3])
```

When creating an empty set, be sure to not use the curly braces `{}` or you will get an empty dictionary instead.

```
1 >>> s = []
2 >>> type(s)
3 <class 'dict'>
```

sets: unordered collections of unique elements

A set automatically remove all the duplicate values.

```
1 >>> s = {1, 2, 3, 2, 3, 4}
2 >>> s
3 {1, 2, 3, 4}
```

And as an unordered data type, they can't be indexed.

```
1 >>> s = {1, 2, 3}
2 >>> s[0]
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 TypeError: 'set' object does not support indexing
6 >>>
```

set add() and update()

Using the `add()` method we can add a single element to the set.

```
1 >>> s = {1, 2, 3}
2 >>> s.add(4)
3 >>> s
4 {1, 2, 3, 4}
```

And with `update()`, multiple ones .

```
1 >>> s = {1, 2, 3}
2 >>> s.update([2, 3, 4, 5, 6])
3 >>> s
4 {1, 2, 3, 4, 5, 6} # remember, sets automatically remove duplicates
```

set remove() and discard()

Both methods will remove an element from the set, but `remove()` will raise a `key_error` if the value doesn't exist.

```
1 >>> s = {1, 2, 3}
2 >>> s.remove(3)
3 >>> s
4 {1, 2}
5 >>> s.remove(3)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 KeyError: 3
```

`discard()` won't raise any errors.

```
1 >>> s = {1, 2, 3}
2 >>> s.discard(3)
3 >>> s
4 {1, 2}
5 >>> s.discard(3)
6 >>>
```

set union()

`union()` or `|` will create a new set that contains all the elements from the sets provided.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {3, 4, 5}
3 >>> s1.union(s2) # or 's1 | s2'
4 {1, 2, 3, 4, 5}
```

set intersection

`intersection` or `&` will return a set containing only the elements that are common to all of them.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s3 = {3, 4, 5}
4 >>> s1.intersection(s2, s3) # or 's1 & s2 & s3'
5 {3}
```

set difference

`difference` or `-` will return only the elements that are unique to the first set (invoked set).

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s1.difference(s2) # or 's1 - s2'
4 {1}
5 >>> s2.difference(s1) # or 's2 - s1'
6 {4}
```

set symmetric_difference

`symmetric_difference` or `^` will return all the elements that are not common between them.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s1.symmetric_difference(s2) # or 's1 ^ s2'
4 {1, 4}
```

itertools Module

The `itertools` module is a collection of tools intended to be fast and use memory efficiently when handling iterators (like [lists](#) or [dictionaries](#)).

From the official [Python 3.x documentation](#):

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

The `itertools` module comes in the standard library and must be imported.

The `operator` module will also be used. This module is not necessary when using `itertools`, but needed for some of the examples below.

accumulate()

Makes an iterator that returns the results of a function.

```
itertools.accumulate(iterable[, func])
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5]
2 >>> result = itertools.accumulate(data, operator.mul)
3 >>> for each in result:
4 >>>     print(each)
5 1
6 2
7 6
8 24
9 120
```

The operator.mul takes two numbers and multiplies them:

```
1 operator.mul(1, 2)
2 2
3 operator.mul(2, 3)
4 6
5 operator.mul(6, 4)
6 24
7 operator.mul(24, 5)
8 120
```

Passing a function is optional:

```
1 >>> data = [5, 2, 6, 4, 5, 9, 1]
2 >>> result = itertools.accumulate(data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 7
7 13
8 17
9 22
10 31
11 32
```

If no function is designated the items will be summed:

```
1 5
2 5 + 2 = 7
3 7 + 6 = 13
4 13 + 4 = 17
5 17 + 5 = 22
6 22 + 9 = 31
7 31 + 1 = 32
```

combinations()

Takes an iterable and a integer. This will create all the unique combination that have r members.

```
itertools.combinations(iterable, r)
```

Example:

```
1 >>> shapes = ['circle', 'triangle', 'square',]
2 >>> result = itertools.combinations(shapes, 2)
3 >>> for each in result:
4 >>>     print(each)
5 ('circle', 'triangle')
6 ('circle', 'square')
7 ('triangle', 'square')
```

combinations_with_replacement()

Just like combinations(), but allows individual elements to be repeated more than once.

```
itertools.combinations_with_replacement(iterable, r)
```

Example:

```
1 >>> shapes = ['circle', 'triangle', 'square']
2 >>> result = itertools.combinations_with_replacement(shapes, 2)
3 >>> for each in result:
4 >>>     print(each)
5 ('circle', 'circle')
6 ('circle', 'triangle')
7 ('circle', 'square')
8 ('triangle', 'triangle')
9 ('triangle', 'square')
10 ('square', 'square')
```

count()

Makes an iterator that returns evenly spaced values starting with number start.

```
itertools.count(start=0, step=1)
```

Example:

```
1 >>> for i in itertools.count(10,3):
2 >>>     print(i)
3 >>>     if i > 20:
4 >>>         break
5 10
6 13
7 16
8 19
9 22
```

cycle()

This function cycles through an iterator endlessly.

```
itertools.cycle(iterable)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet']
2 >>> for color in itertools.cycle(colors):
3 >>>     print(color)
4 red
5 orange
6 yellow
7 green
8 blue
9 violet
10 red
11 orange
```

When reached the end of the iterable it start over again from the beginning.

chain()

Take a series of iterables and return them as one long iterable.

```
itertools.chain(*iterables)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> shapes = ['circle', 'triangle', 'square', 'pentagon']
3 >>> result = itertools.chain(colors, shapes)
4 >>> for each in result:
5 >>>     print(each)
6 red
7 orange
8 yellow
9 green
10 blue
11 circle
12 triangle
13 square
14 pentagon
```

compress()

Filters one iterable with another.

```
itertools.compress(data, selectors)
```

Example:

```
1 >>> shapes = ['circle', 'triangle', 'square', 'pentagon']
2 >>> selections = [True, False, True, False]
3 >>> result = itertools.compress(shapes, selections)
4 >>> for each in result:
5 >>>     print(each)
6 circle
7 square
```

dropwhile()

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.

```
itertools.dropwhile(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.dropwhile(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 6
7 7
8 8
9 9
10 10
11 1
```

filterfalse()

Makes an iterator that filters elements from iterable returning only those for which the predicate is False.

```
itertools.filterfalse(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.filterfalse(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 6
7 7
8 8
9 9
10 10
```

groupby()

Simply put, this function groups things together.

```
itertools.groupby(iterable, key=None)
```

Example:

```
1 >>> robots = [{  
2     'name': 'blaster',  
3     'faction': 'autobot'  
4 }, {  
5     'name': 'galvatron',  
6     'faction': 'decepticon'  
7 }, {  
8     'name': 'jazz',  
9     'faction': 'autobot'  
10 }, {  
11     'name': 'metroplex',  
12     'faction': 'autobot'  
13 }, {  
14     'name': 'megatron',  
15     'faction': 'decepticon'  
16 }, {  
17     'name': 'starcream',  
18     'faction': 'decepticon'  
19 }]  
20 >>> for key, group in itertools.groupby(robots, key=lambda x: x['faction']):  
21 >>>     print(key)  
22 >>>     print(list(group))  
23 autobot  
24 [ {'name': 'blaster', 'faction': 'autobot'}]  
25 decepticon  
26 [ {'name': 'galvatron', 'faction': 'decepticon'}]  
27 autobot  
28 [ {'name': 'jazz', 'faction': 'autobot'}, { 'name': 'metroplex', 'faction': 'autobot'}]  
29 decepticon  
30 [ {'name': 'megatron', 'faction': 'decepticon'}, { 'name': 'starcream', 'faction': 'decepticon'}]
```

islice()

This function is very much like slices. This allows you to cut out a piece of an iterable.

```
itertools.islice(iterable, start, stop[, step])
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]  
2 >>> few_colors = itertools.islice(colors, 2)  
3 >>> for each in few_colors:  
4 >>>     print(each)  
5 red  
6 orange
```

permutations()

```
itertools.permutations(iterable, r=None)
```

Example:

```
1 >>> alpha_data = ['a', 'b', 'c']
2 >>> result = itertools.permutations(alpha_data)
3 >>> for each in result:
4 >>>     print(each)
5 ('a', 'b', 'c')
6 ('a', 'c', 'b')
7 ('b', 'a', 'c')
8 ('b', 'c', 'a')
9 ('c', 'a', 'b')
10 ('c', 'b', 'a')
```

product()

Creates the cartesian products from a series of iterables.

```
1 >>> num_data = [1, 2, 3]
2 >>> alpha_data = ['a', 'b', 'c']
3 >>> result = itertools.product(num_data, alpha_data)
4 >>> for each in result:
5     print(each)
6 (1, 'a')
7 (1, 'b')
8 (1, 'c')
9 (2, 'a')
10 (2, 'b')
11 (2, 'c')
12 (3, 'a')
13 (3, 'b')
14 (3, 'c')
```

repeat()

This function will repeat an object over and over again. Unless, there is a times argument.

```
itertools.repeat(object[, times])
```

Example:

```
1 >>> for i in itertools.repeat("spam", 3):
2     print(i)
3 spam
4 spam
5 spam
```

starmap()

Makes an iterator that computes the function using arguments obtained from the iterable.

```
itertools.starmap(function, iterable)
```

Example:

```
1 >>> data = [(2, 6), (8, 4), (7, 3)]
2 >>> result = itertools.starmap(operator.mul, data)
3 >>> for each in result:
4 >>>     print(each)
5 12
6 32
7 21
```

takewhile()

The opposite of dropwhile(). Makes an iterator and returns elements from the iterable as long as the predicate is true.

```
itertools.takewhile(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.takewhile(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 1
6 2
7 3
8 4
```

tee()

Return n independent iterators from a single iterable.

```
itertools.tee(iterable, n=2)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> alpha_colors, beta_colors = itertools.tee(colors)
3 >>> for each in alpha_colors:
4 >>>     print(each)
5 red
6 orange
7 yellow
8 green
9 blue
```

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> alpha_colors, beta_colors = itertools.tee(colors)
3 >>> for each in beta_colors:
4 >>>     print(each)
5 red
6 orange
7 yellow
8 green
9 blue
```

`zip_longest()`

Makes an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with `fillvalue`. Iteration continues until the longest iterable is exhausted.

```
itertools.zip_longest(*iterables, fillvalue=None)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]
2 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,]
3 >>> for each in itertools.zip_longest(colors, data, fillvalue=None):
4     >>>     print(each)
5     ('red', 1)
6     ('orange', 2)
7     ('yellow', 3)
8     ('green', 4)
9     ('blue', 5)
10    (None, 6)
11    (None, 7)
12    (None, 8)
13    (None, 9)
14    (None, 10)
```

Comprehensions

List comprehension

```
1 >>> a = [1, 3, 5, 7, 9, 11]
2
3 >>> [i - 1 for i in a]
4 [0, 2, 4, 6, 8, 10]
```

Set comprehension

```
1 >>> b = {"abc", "def"}
2 >>> {s.upper() for s in b}
3 {"ABC", "DEF"}
```

Dict comprehension

```
1 >>> c = {'name': 'Pooka', 'age': 5}
2 >>> {v: k for k, v in c.items()}
3 {'Pooka': 'name', 5: 'age'}
```

A List comprehension can be generated from a dictionary:

```
1 >>> c = {'name': 'Pooka', 'first_name': 'Oooka'}
2 >>> ["{}:{}".format(k.upper(), v.upper()) for k, v in c.items()]
3 ['NAME:POOKA', 'FIRST_NAME:OOOKA']
```

Manipulating Strings

Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\\	Backslash

Example:

```
1 >>> print("Hello there!\nHow are you?\nI'm doing fine.")
2 Hello there!
3 How are you?
4 I'm doing fine.
```

Raw Strings

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```
1 >>> print(r'That is Carol\'s cat.')
2 That is Carol\''s cat.
```

Note: mostly used for regular expression definition (see `re` package)

Multiline Strings with Triple Quotes

```
1 >>> print('''Dear Alice,
2 >>>
3 >>> Eve's cat has been arrested for catnapping, cat burglary, and extortion.
4 >>>
5 >>> Sincerely,
6 >>> Bob'''')
7 Dear Alice,
8
9 Eve's cat has been arrested for catnapping, cat burglary, and extortion.
10
11 Sincerely,
12 Bob
```

To keep a nicer flow in your code, you can use the `dedent` function from the `textwrap` standard package.

```
1 >>> from textwrap import dedent
2 >>>
3 >>> def my_function():
4 >>>     print('''
5 >>>         Dear Alice,
6 >>>
```

```
7 >>>      Eve's cat has been arrested for catnapping, cat burglary, and extortion.  
8 >>>  
9 >>>      Sincerely,  
10 >>>      Bob  
11 >>>      ''').strip()
```

This generates the same string than before.

Indexing and Slicing Strings

```
1 H   e   l   l   o       w   o   r   l   d   !  
2 0   1   2   3   4       5   6   7   8   9   10  11
```

```
1 >>> spam = 'Hello world!'  
2  
3 >>> spam[0]  
4 'H'
```

```
1 >>> spam[4]  
2 'o'
```

```
1 >>> spam[-1]  
2 '!'
```

Slicing:

```
1  
2 >>> spam[0:5]  
3 'Hello'
```

```
1 >>> spam[:5]  
2 'Hello'
```

```
1 >>> spam[6:]  
2 'world!'
```

```
1 >>> spam[6:-1]  
2 'world'
```

```
1 >>> spam[:-1]  
2 'Hello world'
```

```
1 >>> spam[::-1]
2 '!dlrow olleH'
```

```
1 >>> spam = 'Hello world!'
2 >>> fizz = spam[0:5]
3 >>> fizz
4 'Hello'
```

The in and not in Operators with Strings

```
1 >>> 'Hello' in 'Hello World'
2 True
```

```
1 >>> 'Hello' in 'Hello'
2 True
```

```
1 >>> 'HELLO' in 'Hello World'
2 False
```

```
1 >>> '' in 'spam'
2 True
```

```
1 >>> 'cats' not in 'cats and dogs'
2 False
```

The in and not in Operators with list

```
1 >>> a = [1, 2, 3, 4]
2 >>> 5 in a
3 False
```

```
1 >>> 2 in a
2 True
```

The upper(), lower(), isupper(), and islower() String Methods

upper() and lower() :

```
1 >>> spam = 'Hello world!'
2 >>> spam = spam.upper()
3 >>> spam
4 'HELLO WORLD!'
```

```
1 >>> spam = spam.lower()
2 >>> spam
3 'hello world!'
```

isupper() and islower():

```
1 >>> spam = 'Hello world!'
2 >>> spam.islower()
3 False
```

```
1 >>> spam.isupper()
2 False
```

```
1 >>> 'HELLO'.isupper()
2 True
```

```
1 >>> 'abc12345'.islower()
2 True
```

```
1 >>> '12345'.islower()
2 False
```

```
1 >>> '12345'.isupper()
2 False
```

The isX String Methods

- **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 new-lines 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.

The startswith() and endswith() String Methods

```
1 >>> 'Hello world!'.startswith('Hello')
2 True
```

```
1 >>> 'Hello world!'.endswith('world!')
2 True
```

```
1 >>> 'abc123'.startswith('abcdef')
2 False
```

```
1 >>> 'abc123'.endswith('12')
2 False
```

```
1 >>> 'Hello world!'.startswith('Hello world!')
2 True
```

```
1 >>> 'Hello world!'.endswith('Hello world!')
2 True
```

The join() and split() String Methods

join():

```
1 >>> ', '.join(['cats', 'rats', 'bats'])
2 'cats, rats, bats'
```

```
1 >>> ' '.join(['My', 'name', 'is', 'Simon'])
2 'My name is Simon'
```

```
1 >>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
2 'MyABCnameABCisABCSimon'
```

split():

```
1 >>> 'My name is Simon'.split()
2 ['My', 'name', 'is', 'Simon']
```

```
1 >>> 'MyABCnameABCisABCSimon'.split('ABC')
2 ['My', 'name', 'is', 'Simon']
```

```
1 >>> 'My name is Simon'.split('m')
2 ['My na', 'e is Si', 'on']
```

Justifying Text with rjust(), ljust(), and center()

rjust() and ljust():

```
1 >>> 'Hello'.rjust(10)
2     Hello'
```

```
1 >>> 'Hello'.rjust(20)
2     Hello'
```

```
1 >>> 'Hello World'.rjust(20)
2     Hello World'
```

```
1 >>> 'Hello'.ljust(10)
2 'Hello      '
```

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
1 >>> 'Hello'.rjust(20, '*')
2 *****Hello*****'
```

```
1 >>> 'Hello'.ljust(20, '-')
2 'Hello-----'
```

`center()`:

```
1 >>> 'Hello'.center(20)
2     Hello      '
```

```
1 >>> 'Hello'.center(20, '=')
2 =====Hello====='
```

Removing Whitespace with `strip()`, `rstrip()`, and `lstrip()`

```
1 >>> spam = '    Hello World    '
2 >>> spam.strip()
3 'Hello World'
```

```
1 >>> spam.lstrip()
2 'Hello World '
```

```
1 >>> spam.rstrip()
2 '    Hello World'
```

```
1 >>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
2 >>> spam.strip('ampS')
3 'BaconSpamEggs'
```

Copying and Pasting Strings with the pyperclip Module (need pip install)

```
1 >>> import pyperclip
2
3 >>> pyperclip.copy('Hello world!')
4
5 >>> pyperclip.paste()
6 'Hello world!'
```

String Formatting

% operator

```
1 >>> name = 'Pete'
2 >>> 'Hello %s' % name
3 "Hello Pete"
```

We can use the `%x` format specifier to convert an int value to a string:

```
1 >>> num = 5
2 >>> 'I have %x apples' % num
3 "I have 5 apples"
```

Note: For new code, using `str.format` or `f-strings` (Python 3.6+) is strongly recommended over the `%` operator.

String Formatting (`str.format`)

Python 3 introduced a new way to do string formatting that was later back-ported to Python 2.7. This makes the syntax for string formatting more regular.

```
1 >>> name = 'John'
2 >>> age = 20
3
4 >>> "Hello I'm {}, my age is {}".format(name, age)
5 "Hello I'm John, my age is 20"
```

```
1 >>> "Hello I'm {0}, my age is {1}".format(name, age)
2 "Hello I'm John, my age is 20"
```

The official [Python 3.x documentation](#) recommend `str.format` over the `%` operator:

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the `str.format()` interface helps avoid these errors. These

alternatives also provide more powerful, flexible and extensible approaches to formatting text.

Lazy string formatting

You would only use `%s` string formatting on functions that can do lazy parameters evaluation, the most common being logging:

Prefer:

```
1 >>> name = "alice"
2 >>> logging.debug("User name: %s", name)
```

Over:

```
>>> logging.debug("User name: {}".format(name))
```

Or:

```
>>> logging.debug("User name: " + name)
```

Formatted String Literals or f-strings (Python 3.6+)

```
1 >>> name = 'Elizabeth'
2 >>> f'Hello {name}!'
3 'Hello Elizabeth!'
```

It is even possible to do inline arithmetic with it:

```
1 >>> a = 5
2 >>> b = 10
3 >>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
4 'Five plus ten is 15 and not 30.'
```

Template Strings

A simpler and less powerful mechanism, but it is recommended when handling format strings generated by users. Due to their reduced complexity template strings are a safer choice.

```
1 >>> from string import Template
2 >>> name = 'Elizabeth'
3 >>> t = Template('Hey $name!')
4 >>> t.substitute(name=name)
5 'Hey Elizabeth!'
```

Regular Expressions

1. Import the regex module with `import re`.

2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a `Match` object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

All the regex functions in Python are in the `re` module:

```
>>> import re
```

Matching Regex Objects

```
1 >>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
2
3 >>> mo = phone_num_regex.search('My number is 415-555-4242.')
4
5 >>> print('Phone number found: {}'.format(mo.group()))
6 Phone number found: 415-555-4242
```

Grouping with Parentheses

```
1 >>> phone_num_regex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
2
3 >>> mo = phone_num_regex.search('My number is 415-555-4242.')
4
5 >>> mo.group(1)
6 '415'
7
8 >>> mo.group(2)
9 '555-4242'
10
11 >>> mo.group(0)
12 '415-555-4242'
13
14 >>> mo.group()
15 '415-555-4242'
```

To retrieve all the groups at once: use the `groups()` method—note the plural form for the name.

```
1 >>> mo.groups()
2 ('415', '555-4242')
3
4 >>> area_code, main_number = mo.groups()
5
6 >>> print(area_code)
7 415
8
9 >>> print(main_number)
10 555-4242
```

Matching Multiple Groups with the Pipe

The `|` character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

```
1 >>> hero_regex = re.compile (r'Batman|Tina Fey')
```

```
3 >>> mo1 = hero_regex.search('Batman and Tina Fey.')
4
5 >>> mo1.group()
6 'Batman'
7
8 >>> mo2 = hero_regex.search('Tina Fey and Batman.')
9
10 >>> mo2.group()
11 'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex:

```
1 >>> bat_regex = re.compile(r'Bat(man|mobile|copter|bat)')
2
3 >>> mo = bat_regex.search('Batmobile lost a wheel')
4
5 >>> mo.group()
6 'Batmobile'
7
8 >>> mo.group(1)
9 'mobile'
```

Optional Matching with the Question Mark

The ? character flags the group that precedes it as an optional part of the pattern.

```
1 >>> bat_regex = re.compile(r'Bat(wo)?man')
2 >>> mo1 = bat_regex.search('The Adventures of Batman')
3 >>> mo1.group()
4 'Batman'
5
6 >>> mo2 = bat_regex.search('The Adventures of Batwoman')
7 >>> mo2.group()
8 'Batwoman'
```

Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text.

```
1 >>> bat_regex = re.compile(r'Bat(wo)*man')
2 >>> mo1 = bat_regex.search('The Adventures of Batman')
3 >>> mo1.group()
4 'Batman'
5
6 >>> mo2 = bat_regex.search('The Adventures of Batwoman')
7 >>> mo2.group()
8 'Batwoman'
9
10 >>> mo3 = bat_regex.search('The Adventures of Batwowowowoman')
11 >>> mo3.group()
12 'Batwowowowoman'
```

Matching One or More with the Plus

While * means “match zero or more,” the + (or plus) means “match one or more”. The group preceding a plus must appear at least once. It is not optional:

```
1 >>> bat_regex = re.compile(r'Bat(wo)+man')
2 >>> mo1 = bat_regex.search('The Adventures of Batwoman')
3 >>> mo1.group()
4 'Batwoman'
```

```
1 >>> mo2 = bat_regex.search('The Adventures of Batwowowowoman')
2 >>> mo2.group()
3 'Batwowowowoman'
```

```
1 >>> mo3 = bat_regex.search('The Adventures of Batman')
2 >>> mo3 is None
3 True
```

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex `(Ha){3}` will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

```
1 >>> ha_regex = re.compile(r'(Ha){3}')
2 >>> mo1 = ha_regex.search('HaHaHa')
3 >>> mo1.group()
4 'HaHaHa'
```

```
1 >>> mo2 = ha_regex.search('Ha')
2 >>> mo2 is None
3 True
```

Greedy and Nongreedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
1 >>> greedy_ha_regex = re.compile(r'(Ha){3,5}')
2 >>> mo1 = greedy_ha_regex.search('HaHaHaHaHa')
3 >>> mo1.group()
4 'HaHaHaHaHa'
```

```
1 >>> nongreedy_ha_regex = re.compile(r'(Ha){3,5}?' )
2 >>> mo2 = nongreedy_ha_regex.search('HaHaHaHaHa')
3 >>> mo2.group()
4 'HaHaHa'
```

The `findall()` Method

In addition to the `search()` method, Regex objects also have a `findall()` method. While `search()` will return a `Match` object of the first matched text in the searched string, the `findall()` method will return the strings of every match in the searched string.

```
1 >>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d') # has no groups
2
3 >>> phone_num_regex.findall('Cell: 415-555-9999 Work: 212-555-0000')
4 ['415-555-9999', '212-555-0000']
```

To summarize what the `findall()` method returns, remember the following:

- When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d`, the method `findall()` returns a list of strings, such as `['415-555-9999', '212-555-0000']`.
- When called on a regex that has groups, such as `(\d\d\d)-(\d\d)-(\d\d\d)`, the method `findall()` returns a list of lists of strings (one string for each group), such as `[['415', '555', '9999'], ['212', '555', '0000']]`.

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase.

```
1 >>> vowel_regex = re.compile(r'[aeiouAEIOU]')
2
3 >>> vowel_regex.findall('Robocop eats baby food. BABY FOOD.')
4 ['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class. For example, enter the following into the interactive shell:

```
1 >>> consonant_regex = re.compile(r'[^aeiouAEIOU]')
2
3 >>> consonant_regex.findall('Robocop eats baby food. BABY FOOD.')
4 ['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', ' ', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

The Caret and Dollar Sign Characters

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign (\$) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and \\$ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

The `r'^Hello'` regular expression string matches strings that begin with 'Hello':

```
1 >>> begins_with_hello = re.compile(r'^Hello')
2
3 >>> begins_with_hello.search('Hello world!')
4 <sre.SRE_Match object; span=(0, 5), match='Hello'>
5
```

```
6 >>> begins_with_hello.search('He said hello.') is None
7 True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9:

```
1 >>> whole_string_is_num = re.compile(r'^\d+$')
2
3 >>> whole_string_is_num.search('1234567890')
4 <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
5
6 >>> whole_string_is_num.search('12345xyz67890') is None
7 True
8
9 >>> whole_string_is_num.search('12 34567890') is None
10 True
```

The Wildcard Character

The `.` (or dot) character in a regular expression is called a wildcard and will match any character except for a newline:

```
1 >>> at_regex = re.compile(r'.at')
2
3 >>> at_regex.findall('The cat in the hat sat on the flat mat.')
4 ['cat', 'hat', 'sat', 'lat', 'mat']
```

Matching Everything with Dot-Star

```
1 >>> name_regex = re.compile(r'First Name: (.*) Last Name: (.*)')
2
3 >>> mo = name_regex.search('First Name: Al Last Name: Sweigart')
4
5 >>> mo.group(1)
6 'Al'
```

```
1 >>> mo.group(2)
2 'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark `(.*?)`. The question mark tells Python to match in a nongreedy way:

```
1 >>> nongreedy_regex = re.compile(r'<.*?>')
2 >>> mo = nongreedy_regex.search('<To serve man> for dinner.>')
3 >>> mo.group()
4 '<To serve man>'
```

```
1 >>> greedy_regex = re.compile(r'<.*>')
2 >>> mo = greedy_regex.search('<To serve man> for dinner.>')
3 >>> mo.group()
4 '<To serve man> for dinner.>'
```

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character:

```
1 >>> no_newline_regex = re.compile('.*')
2 >>> no_newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
3 'Serve the public trust.'
```

```
1 >>> newline_regex = re.compile('.*', re.DOTALL)
2 >>> newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
3 'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

Review of Regex Symbols

Symbol	Matches
?	zero or one of the preceding group.
*	zero or more of the preceding group.
+	one or more of the preceding group.
{n}	exactly n of the preceding group.
{n,}	n or more of the preceding group.
{,m}	0 to m of the preceding group.
{n,m}	at least n and at most m of the preceding p.
{n,m}? or *? or +?	performs a nongreedy match of the preceding p.
^spam	means the string must begin with spam.
spam\$	means the string must end with spam.
.	any character, except newline characters.
\d , \w , and \s	a digit, word, or space character, respectively.
\D , \W , and \S	anything except a digit, word, or space, respectively.
[abc]	any character between the brackets (such as a, b,).
[^abc]	any character that isn't between the brackets.

Case-Insensitive Matching

To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile():

```
1 >>> robocop = re.compile(r'robocop', re.I)
2
3 >>> robocop.search('Robocop is part man, part machine, all cop.').group()
4 'Robocop'
```

```
1 >>> robocop.search('ROBOCOP protects the innocent.').group()
2 'ROBOCOP'
```

```
1 >>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
2 'robocop'
```

Substituting Strings with the sub() Method

The sub() method for Regex objects is passed two arguments:

1. The first argument is a string to replace any matches.
2. The second is the string for the regular expression.

The sub() method returns a string with the substitutions applied:

```
1 >>> names_regex = re.compile(r'Agent \w+')
2
3 >>> names_regex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
4 'CENSORED gave the secret documents to CENSORED.'
```

Another example:

```
1 >>> agent_names_regex = re.compile(r'Agent (\w)\w*')
2
3 >>> agent_names_regex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent')
4 ***** told **** that **** knew ***** was a double agent.'
```

Managing Complex Regexes

To tell the re.compile() function to ignore whitespace and comments inside the regular expression string, “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

Now instead of a hard-to-read regular expression like this:

```
phone_regex = re.compile(r'((\d{3})|(\d{3}))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s*(ext|x|ext.)\s*\d{2,5})?')
```

you can spread the regular expression over multiple lines with comments like this:

```
1 phone_regex = re.compile(r'''(
2     (\d{3})|(\d{3})?
3     # area code
4     (\s|-|\.)?
5     # separator
6     \d{3}
7     # first 3 digits
8     (\s|-|\.)
9     \d{4}
10    # separator
11    (\s*(ext|x|ext.)\s*\d{2,5})?
12    # last 4 digits
13    # extension
14 )''', re.VERBOSE)
```

Handling File and Directory Paths

There are two main modules in Python that deals with path manipulation. One is the `os.path` module and the other is the `pathlib` module. The `pathlib` module was added in Python 3.4, offering an object-oriented way to handle file system paths.

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (`\`) as the separator between folder names. On Unix based operating system such as macOS, Linux, and BSDs, the forward slash (`/`) is used as the path separator. Joining paths can be a headache if your code needs to work on different platforms.

Fortunately, Python provides easy ways to handle this. We will showcase how to deal with this with both `os.path.join` and `pathlib.Path.joinpath`

Using `os.path.join` on Windows:

```
1 >>> import os
2
3 >>> os.path.join('usr', 'bin', 'spam')
4 'usr\\bin\\spam'
```

And using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2
3 >>> print(Path('usr').joinpath('bin').joinpath('spam'))
4 usr/bin/spam
```

`pathlib` also provides a shortcut to joinpath using the `/` operator:

```
1 >>> from pathlib import Path
2
3 >>> print(Path('usr') / 'bin' / 'spam')
4 usr/bin/spam
```

Notice the path separator is different between Windows and Unix based operating system, that's why you want to use one of the above methods instead of adding strings together to join paths together.

Joining paths is helpful if you need to create different file paths under the same directory.

Using `os.path.join` on Windows:

```
1 >>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
2
3 >>> for filename in my_files:
4     >>>     print(os.path.join('C:\\Users\\asweigart', filename))
5 C:\\Users\\asweigart\\accounts.txt
6 C:\\Users\\asweigart\\details.csv
7 C:\\Users\\asweigart\\invite.docx
```

Using `pathlib` on *nix:

```
1 >>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
2 >>> home = Path.home()
3 >>> for filename in my_files:
4 >>>     print(home / filename)
5 /home/asweigart/accounts.txt
6 /home/asweigart/details.csv
7 /home/asweigart/invite.docx
```

The Current Working Directory

Using `os` on Windows:

```
1 >>> import os
2
3 >>> os.getcwd()
4 'C:\\Python34'
5 >>> os.chdir('C:\\Windows\\System32')
6
7 >>> os.getcwd()
8 'C:\\Windows\\System32'
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> from os import chdir
3
4 >>> print(Path.cwd())
5 /home/asweigart
6
7 >>> chdir('/usr/lib/python3.6')
8 >>> print(Path.cwd())
9 /usr/lib/python3.6
```

Creating New Folders

Using `os` on Windows:

```
1 >>> import os
2 >>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> cwd = Path.cwd()
3 >>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir()
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "/usr/lib/python3.6/pathlib.py", line 1226, in mkdir
7     self._accessor.mkdir(self, mode)
8   File "/usr/lib/python3.6/pathlib.py", line 387, in wrapped
9     return strfunc(str(pathobj), *args)
10 FileNotFoundError: [Errno 2] No such file or directory: '/home/asweigart/delicious/walnut/waffles'
```

Oh no, we got a nasty error! The reason is that the 'delicious' directory does not exist, so we cannot make the 'walnut' and the 'waffles' directories under it. To fix this, do:

```
1 >>> from pathlib import Path
2 >>> cwd = Path.cwd()
3 >>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir(parents=True)
```

And all is good :)

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's current working directory

There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

Handling Absolute and Relative Paths

To see if a path is an absolute path:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isabs('/')
3 True
4 >>> os.path.isabs('..')
5 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('/').is_absolute()
3 True
4 >>> Path('..').is_absolute()
5 False
```

You can extract an absolute path with both `os.path` and `pathlib`

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.getcwd()
3 '/home/asweigart'
4 >>> os.path.abspath('..')
5 '/home'
```

Using `pathlib` on *nix:

```
1 from pathlib import Path
2 print(Path.cwd())
3 /home/asweigart
4 print(Path('..').resolve())
5 /home
```

You can get a relative path from a starting path to another path.

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.relpath('/etc/passwd', '/')
3 'etc/passwd'
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> print(Path('/etc/passwd').relative_to('/'))
3 etc/passwd
```

Checking Path Validity

Checking if a file/directory exists:

Using `os.path` on *nix:

```
1 import os
2 >>> os.path.exists('.')
3 True
4 >>> os.path.exists('setup.py')
5 True
6 >>> os.path.exists('/etc')
7 True
8 >>> os.path.exists('nonexistentfile')
9 False
```

Using `pathlib` on *nix:

```
1 from pathlib import Path
2 >>> Path('.').exists()
3 True
4 >>> Path('setup.py').exists()
5 True
6 >>> Path('/etc').exists()
7 True
8 >>> Path('nonexistentfile').exists()
9 False
```

Checking if a path is a file:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isfile('setup.py')
3 True
4 >>> os.path.isfile('/home')
5 False
6 >>> os.path.isfile('nonexistentfile')
7 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('setup.py').is_file()
3 True
4 >>> Path('/home').is_file()
5 False
6 >>> Path('nonexistentfile').is_file()
7 False
```

Checking if a path is a directory:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isdir('/')
3 True
4 >>> os.path.isdir('setup.py')
5 False
6 >>> os.path.isdir('/spam')
7 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('/').is_dir()
3 True
4 >>> Path('setup.py').is_dir()
5 False
6 >>> Path('/spam').is_dir()
7 False
```

Finding File Sizes and Folder Contents

Getting a file's size in bytes:

Using `os.path` on Windows:

```
1 >>> import os
2 >>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
3 776192
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> stat = Path('/bin/python3.6').stat()
3 >>> print(stat) # stat contains some other information about the file as well
4 os.stat_result(st_mode=33261, st_ino=141087, st_dev=2051, st_nlink=2, st_uid=0,
5 --snip--
6 st_gid=0, st_size=10024, st_atime=1517725562, st_mtime=1515119809, st_ctime=1517261276)
7 >>> print(stat.st_size) # size in bytes
8 10024
```

Listing directory contents using `os.listdir` on Windows:

```
1 >>> import os
2 >>> os.listdir('C:\\Windows\\System32')
3 ['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
4 --snip--
5 'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Listing directory contents using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> for f in Path('/usr/bin').iterdir():
3 >>>     print(f)
4 ...
5 /usr/bin/tiff2rgba
6 /usr/bin/iconv
7 /usr/bin/ldd
8 /usr/bin/cache_restore
9 /usr/bin/udiskie
10 /usr/bin/unix2dos
11 /usr/bin/tlreencode
12 /usr/bin/epstopdf
13 /usr/bin/idle3
14 ...
```

To find the total size of all the files in this directory:

WARNING: Directories themselves also have a size! So you might want to check for whether a path is a file or directory using the methods discussed in the above section!

Using `os.path.getsize()` and `os.listdir()` together on Windows:

```
1 >>> import os
2 >>> total_size = 0
3
4 >>> for filename in os.listdir('C:\\Windows\\System32'):
5     total_size = total_size + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
6
7 >>> print(total_size)
8 1117846456
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> total_size = 0
3
4 >>> for sub_path in Path('/usr/bin').iterdir():
5 ...     total_size += sub_path.stat().st_size
```

```
6 >>>
7 >>> print(total_size)
8 1903178911
```

Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders.

```
1 >>> import shutil, os
2
3 >>> os.chdir('C:\\\\')
4
5 >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
6   'C:\\delicious\\\\spam.txt'
7
8 >>> shutil.copy('eggs.txt', 'C:\\delicious\\\\eggs2.txt')
9   'C:\\delicious\\\\eggs2.txt'
```

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it:

```
1 >>> import shutil, os
2
3 >>> os.chdir('C:\\\\')
4
5 >>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
6   'C:\\bacon_backup'
```

Moving and Renaming Files and Folders

```
1 >>> import shutil
2 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
3   'C:\\eggs\\\\bacon.txt'
```

The destination path can also specify a filename. In the following example, the source file is moved and renamed:

```
1 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\\\new_bacon.txt')
2   'C:\\eggs\\\\new_bacon.txt'
```

If there is no eggs folder, then `move()` will rename bacon.txt to a file named eggs.

```
1 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
2   'C:\\eggs'
```

Permanently Deleting Files and Folders

- Calling `os.unlink(path)` or `Path.unlink()` will delete the file at `path`.
- Calling `os.rmdir(path)` or `Path.rmdir()` will delete the folder at `path`. This folder must be empty of any files or folders.
- Calling `shutil.rmtree(path)` will remove the folder at `path`, and all files and folders it contains will also be deleted.

Safe Deletes with the send2trash Module

You can install this module by running pip install send2trash from a Terminal window.

```
1 >>> import send2trash
2
3 >>> with open('bacon.txt', 'a') as bacon_file: # creates the file
4 ...     bacon_file.write('Bacon is not a vegetable.')
5 25
6
7 >>> send2trash.send2trash('bacon.txt')
```

Walking a Directory Tree

```
1 >>> import os
2 >>>
3 >>> for folder_name, subfolders, filenames in os.walk('C:\\delicious'):
4 >>>     print('The current folder is {}'.format(folder_name))
5 >>>
6 >>>     for subfolder in subfolders:
7 >>>         print('SUBFOLDER OF {}: {}'.format(folder_name, subfolder))
8 >>>     for filename in filenames:
9 >>>         print('FILE INSIDE {}: {}'.format(folder_name, filename))
10 >>>
11 >>>     print('')
12 The current folder is C:\\delicious
13 SUBFOLDER OF C:\\delicious: cats
14 SUBFOLDER OF C:\\delicious: walnut
15 FILE INSIDE C:\\delicious: spam.txt
16
17 The current folder is C:\\delicious\\cats
18 FILE INSIDE C:\\delicious\\cats: catnames.txt
19 FILE INSIDE C:\\delicious\\cats: zophie.jpg
20
21 The current folder is C:\\delicious\\walnut
22 SUBFOLDER OF C:\\delicious\\walnut: waffles
23
24 The current folder is C:\\delicious\\walnut\\waffles
25 FILE INSIDE C:\\delicious\\walnut\\waffles: butter.txt
```

`pathlib` provides a lot more functionality than the ones listed above, like getting file name, getting file extension, reading/writing a file without manually opening it, etc. Check out the [official documentation](#) if you want to know more!

Reading and Writing Files

The File Reading/Writing Process

To read/write to a file in Python, you will want to use the `with` statement, which will close the file for you after you are done.

Opening and reading files with the `open()` function

```
1 >>> with open('C:\\\\Users\\\\your_home_folder\\\\hello.txt') as hello_file:
2 ...     hello_content = hello_file.read()
3 >>> hello_content
4 'Hello World!'
5
6 >>> # Alternatively, you can use the *readlines()* method to get a list of string values from the file, one string
7
8 >>> with open('sonnet29.txt') as sonnet_file:
9 ...     sonnet_file.readlines()
10 [When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
11 outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
12 look upon myself and curse my fate,']
13
```

```
14 >>> # You can also iterate through the file line by line:
15 >>> with open('sonnet29.txt') as sonnet_file:
16 ...     for line in sonnet_file: # note the new line character will be included in the line
17 ...         print(line, end='')
18
19 When, in disgrace with fortune and men's eyes,
20 I all alone beweep my outcast state,
21 And trouble deaf heaven with my bootless cries,
22 And look upon myself and curse my fate,
```

Writing to Files

```
1 >>> with open('bacon.txt', 'w') as bacon_file:
2 ...     bacon_file.write('Hello world!\n')
3 13
4
5 >>> with open('bacon.txt', 'a') as bacon_file:
6 ...     bacon_file.write('Bacon is not a vegetable.')
7 25
8
9 >>> with open('bacon.txt') as bacon_file:
10 ...     content = bacon_file.read()
11
12 >>> print(content)
13 Hello world!
14 Bacon is not a vegetable.
```

Saving Variables with the shelve Module

To save variables:

```
1 >>> import shelve
2
3 >>> cats = ['Zophie', 'Pooka', 'Simon']
4 >>> with shelve.open('mydata') as shelf_file:
5 ...     shelf_file['cats'] = cats
```

To open and read variables:

```
1 >>> with shelve.open('mydata') as shelf_file:
2 ...     print(type(shelf_file))
3 ...     print(shelf_file['cats'])
4 <class 'shelve.DbfilenameShelf'>
5 ['Zophie', 'Pooka', 'Simon']
```

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the `list()` function to get them in list form.

```
1 >>> with shelve.open('mydata') as shelf_file:
2 ...     print(list(shelf_file.keys()))
3 ...     print(list(shelf_file.values()))
4 ['cats']
5 [['Zophie', 'Pooka', 'Simon']]
```

Saving Variables with the pprint.pformat() Function

```
1 >>> import pprint
2
3 >>> cats = [{"name": "Zophie", "desc": "chubby"}, {"name": "Pooka", "desc": "fluffy"}]
4
5 >>> pprint.pformat(cats)
6 "[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
7
8 >>> with open('myCats.py', 'w') as file_obj:
9     ...     file_obj.write('cats = {}\\n'.format(pprint.pformat(cats)))
10    83
```

Reading ZIP Files

```
1 >>> import zipfile, os
2
3 >>> os.chdir('C:\\')      # move to the folder with example.zip
4 >>> with zipfile.ZipFile('example.zip') as example_zip:
5     ...     print(example_zip.namelist())
6     ...     spam_info = example_zip.getinfo('spam.txt')
7     ...     print(spam_info.file_size)
8     ...     print(spam_info.compress_size)
9     ...     print('Compressed file is %sx smaller!' % (round(spam_info.file_size / spam_info.compress_size, 2)))
10
11 ['spam.txt', 'cats\\', 'cats\\catnames.txt', 'cats\\zophie.jpg']
12 13908
13 3828
14 'Compressed file is 3.63x smaller!'
```

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
1 >>> import zipfile, os
2
3 >>> os.chdir('C:\\')      # move to the folder with example.zip
4
5 >>> with zipfile.ZipFile('example.zip') as example_zip:
6     ...     example_zip.extractall()
```

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
1 >>> with zipfile.ZipFile('example.zip') as example_zip:
2     ...     print(example_zip.extract('spam.txt'))
3     ...     print(example_zip.extract('spam.txt', 'C:\\\\some\\\\new\\\\folders'))
4 'C:\\spam.txt'
5 'C:\\some\\new\\folders\\spam.txt'
```

Creating and Adding to ZIP Files

```
1 >>> import zipfile
2
3 >>> with zipfile.ZipFile('new.zip', 'w') as new_zip:
4     ...     new_zip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
```

This code will create a new ZIP file named `new.zip` that has the compressed contents of `spam.txt`.

JSON, YAML and configuration files

JSON

Open a JSON file with:

```
1 import json
2 with open("filename.json", "r") as f:
3     content = json.loads(f.read())
```

Write a JSON file with:

```
1 import json
2
3 content = {"name": "Joe", "age": 20}
4 with open("filename.json", "w") as f:
5     f.write(json.dumps(content, indent=2))
```

YAML

Compared to JSON, YAML allows for much better human maintainability and gives you the option to add comments. It is a convenient choice for configuration files where humans will have to edit it.

There are two main libraries allowing to access to YAML files:

- [PyYaml](#)
- [Ruamel.yaml](#)

Install them using `pip install` in your virtual environment.

The first one is easier to use but the second one, Ruamel, implements much better the YAML specification, and allows for example to modify a YAML content without altering comments.

Open a YAML file with:

```
1 from ruamel.yaml import YAML
2
3 with open("filename.yaml") as f:
4     yaml=YAML()
5     yaml.load(f)
```

Anyconfig

[Anyconfig](#) is a very handy package allowing to abstract completely the underlying configuration file format. It allows to load a Python dictionary from JSON, YAML, TOML, and so on.

Install it with:

```
pip install anyconfig
```

Usage:

```
1 import anyconfig
2
3 conf1 = anyconfig.load("/path/to/foo/conf.d/a.yml")
```

Debugging

Raising Exceptions

Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function

```
1 >>> raise Exception('This is the error message.')
2 Traceback (most recent call last):
3   File "<pyshell#191>", line 1, in <module>
4     raise Exception('This is the error message.')
5 Exception: This is the error message.
```

Often it's the code that calls the function, not the function itself, that knows how to handle an exception. So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

```
1 def box_print(symbol, width, height):
2     if len(symbol) != 1:
3         raise Exception('Symbol must be a single character string.')
4     if width <= 2:
5         raise Exception('Width must be greater than 2.')
6     if height <= 2:
7         raise Exception('Height must be greater than 2.')
8     print(symbol * width)
9     for i in range(height - 2):
10        print(symbol + (' ' * (width - 2)) + symbol)
11     print(symbol * width)
12 for sym, w, h in (('*', 4, 4), ('0', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
13     try:
14         box_print(sym, w, h)
15     except Exception as err:
16         print('An exception happened: ' + str(err))
```

Getting the Traceback as a String

The traceback is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception. You will need to import Python's traceback module before calling this function.

```
1 >>> import traceback
2
3 >>> try:
4 >>>     raise Exception('This is the error message.')
5 >>> except:
6 >>>     with open('errorInfo.txt', 'w') as error_file:
7 >>>         error_file.write(traceback.format_exc())
8 >>>     print('The traceback info was written to errorInfo.txt.')
9 116
10 The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the write() method, since 116 characters were written to the file. The traceback text was written to errorInfo.txt.

```
1 Traceback (most recent call last):
2   File "<pyshell#28>", line 2, in <module>
3     Exception: This is the error message.
```

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

```
1  >>> pod_bay_door_status = 'open'
2
3  >>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
4
5  >>> pod_bay_door_status = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'
6
7  >>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
8
9  Traceback (most recent call last):
10    File "<pyshell#10>", line 1, in <module>
11      assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
12  AssertionError: The pod bay doors need to be "open".
```

In plain English, an assert statement says, “I assert that this condition holds true, and if not, there is a bug somewhere in the program.” Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Disabling Assertions

Assertions can be disabled by passing the -O option when running Python.

Logging

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the #! python shebang line):

```
1 import logging
2
3 logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
```

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is $1 \times 2 \times 3 \times 4$, or 24. Factorial 7 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as factorialLog.py.

```

1  >>> import logging
2  >>>
3  >>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
4  >>>
5  >>> logging.debug('Start of program')
6  >>>
7  >>> def factorial(n):
8  >>>
9  >>>     logging.debug('Start of factorial(%s)' % (n))
10 >>>     total = 1
11 >>>
12 >>>     for i in range(1, n + 1):
13 >>>         total *= i
14 >>>         logging.debug('i is ' + str(i) + ', total is ' + str(total))
15 >>>
16 >>>     logging.debug('End of factorial(%s)' % (n))
17 >>>
18 >>>     return total
19 >>>
20 >>> print(factorial(5))
21 >>> logging.debug('End of program')
22 2015-05-23 16:20:12,664 - DEBUG - Start of program
23 2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
24 2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
25 2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
26 2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
27 2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
28 2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
29 2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
30 2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
31 0
32 2015-05-23 16:20:12,684 - DEBUG - End of program

```

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
DEBUG	<code>logging.debug()</code>	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	<code>logging.info()</code>	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	<code>logging.warning()</code>	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	<code>logging.error()</code>	Used to record an error that caused the program to fail to do something.
CRITICAL	<code>logging.critical()</code>	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand.

```

1  >>> import logging
2
3  >>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s - %(levelname)s- %(message)s')
4
5  >>> logging.critical('Critical error! Critical error!')

```

```
6 2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
7
8 >>> logging.disable(logging.CRITICAL)
9
10 >>> logging.critical('Critical error! Critical error!')
11
12 >>> logging.error('Error! Error!')
```

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
1 import logging
2
3 logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
```

Lambda Functions

This function:

```
1 >>> def add(x, y):
2         return x + y
3
4 >>> add(5, 3)
5 8
```

Is equivalent to the *lambda* function:

```
1 >>> add = lambda x, y: x + y
2 >>> add(5, 3)
3 8
```

It's not even need to bind it to a name like `add` before:

```
1 >>> (lambda x, y: x + y)(5, 3)
2 8
```

Like regular nested functions, lambdas also work as lexical closures:

```
1 >>> def make_adder(n):
2         return lambda x: x + n
3
4 >>> plus_3 = make_adder(3)
5 >>> plus_5 = make_adder(5)
6
7 >>> plus_3(4)
8 7
9 >>> plus_5(4)
10 9
```

Note: lambda can only evaluate an expression, like a single line of code.

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
1 >>> age = 15
2
3 >>> print('kid' if age < 18 else 'adult')
4 kid
```

Ternary operators can be chained:

```
1 >>> age = 15
2
3 >>> print('kid' if age < 13 else 'teenager' if age < 18 else 'adult')
4 teenager
```

The code above is equivalent to:

```
1 if age < 18:
2     if age < 13:
3         print('kid')
4     else:
5         print('teenager')
6 else:
7     print('adult')
```

args and kwargs

The names `args` and `kwargs` are arbitrary - the important thing are the `*` and `**` operators. They can mean:

1. In a function declaration, `*` means “pack all remaining positional arguments into a tuple named `<name>`”, while `**` is the same for keyword arguments (except it uses a dictionary, not a tuple).
2. In a function call, `*` means “unpack tuple or list named `<name>` to positional arguments at this position”, while `**` is the same for keyword arguments.

For example you can make a function that you can use to call any other function, no matter what parameters it has:

```
1 def forward(f, *args, **kwargs):
2     return f(*args, **kwargs)
```

Inside forward, args is a tuple (of all positional arguments except the first one, because we specified it - the f), kwargs is a dict. Then we call f and unpack them so they become normal arguments to f.

You use `*args` when you have an indefinite amount of positional arguments.

```
1 >>> def fruits(*args):
2 >>>     for fruit in args:
3 >>>         print(fruit)
4
5 >>> fruits("apples", "bananas", "grapes")
6
7 "apples"
8 "bananas"
9 "grapes"
```

Similarly, you use `**kwargs` when you have an indefinite number of keyword arguments.

```
1 >>> def fruit(**kwargs):
2 >>>     for key, value in kwargs.items():
3 >>>         print("{0}: {1}".format(key, value))
4
5 >>> fruit(name = "apple", color = "red")
6
7 name: apple
8 color: red
```

```
1 >>> def show(arg1, arg2, *args, kwarg1=None, kwarg2=None, **kwargs):
2 >>>     print(arg1)
3 >>>     print(arg2)
4 >>>     print(args)
5 >>>     print(kwarg1)
6 >>>     print(kwarg2)
7 >>>     print(kwargs)
8
9 >>> data1 = [1,2,3]
10 >>> data2 = [4,5,6]
11 >>> data3 = {'a':7,'b':8,'c':9}
12
13 >>> show(*data1,*data2, kwarg1="python",kwarg2="cheatsheet",**data3)
14 1
15 2
16 (3, 4, 5, 6)
17 python
18 cheatsheet
19 {'a': 7, 'b': 8, 'c': 9}
20
21 >>> show(*data1, *data2, **data3)
22 1
23 2
24 (3, 4, 5, 6)
25 None
26 None
27 {'a': 7, 'b': 8, 'c': 9}
28
29 # If you do not specify ** for kwargs
30 >>> show(*data1, *data2, *data3)
31 1
32 2
33 (3, 4, 5, 6, "a", "b", "c")
34 None
35 None
36 []
```

Things to Remember(args)

1. Functions can accept a variable number of positional arguments by using `*args` in the def statement.
2. You can use the items from a sequence as the positional arguments for a function with the `*` operator.
3. Using the `*` operator with a generator may cause your program to run out of memory and crash.
4. Adding new positional parameters to functions that accept `*args` can introduce hard-to-find bugs.

Things to Remember(kwargs)

1. Function arguments can be specified by position or by keyword.
2. Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
3. Keyword arguments with default values make it easy to add new behaviors to a function, especially when the function has existing callers.
4. Optional keyword arguments should always be passed by keyword instead of by position.

Context Manager

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes.

with statement

A context manager is an object that is notified when a context (a block of code) starts and ends. You commonly use one with the `with` statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
1 >>> with open(filename) as f:  
2   ...     file_contents = f.read()  
3  
4 # the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's `exit` method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way.

Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
1 >>> import contextlib  
2 >>> @contextlib.contextmanager  
3 ... def context_manager(num):  
4 ...     print('Enter')  
5 ...     yield num + 1  
6 ...     print('Exit')  
7 >>> with context_manager(2) as cm:  
8 ...     # the following instructions are run when the 'yield' point of the context  
9 ...     # manager is reached.  
10 ...    # 'cm' will have the value that was yielded  
11 ...    print('Right in the middle with cm = {}'.format(cm))  
12 Enter  
13 Right in the middle with cm = 3  
14 Exit  
15  
16 >>>
```

`__main__` Top-level script environment

`__main__` is the name of the scope in which top-level code executes. A module's `name` is set equal to `__main__` when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run as a script or with `python -m` but not when it is imported:

```
1 >>> if __name__ == "__main__":
2 ...     # execute only if run as a script
3 ...     main()
```

For a package, the same effect can be achieved by including a `main.py` module, the contents of which will be executed when the module is run with `-m`

For example we are developing script which is designed to be used as module, we should do:

```
1 >>> # Python program to execute function directly
2 >>> def add(a, b):
3 ...     return a+b
4 ...
5 >>> add(10, 20) # we can test it by calling the function save it as calculate.py
6 30
7 >>> # Now if we want to use that module by importing we have to comment out our call,
8 >>> # Instead we can write like this in calculate.py
9 >>> if __name__ == "__main__":
10 ...     add(3, 5)
11 ...
12 >>> import calculate
13 >>> calculate.add(3, 5)
14 8
```

Advantages

1. Every Python module has its `__name__` defined and if this is `__main__`, it implies that the module is being run standalone by the user and we can do corresponding appropriate actions.
2. If you import this script as a module in another script, the `name` is set to the name of the script/module.
3. Python files can act as either reusable modules, or as standalone programs.
4. if `__name__ == "main"`: is used to execute some code only if the file was run directly, and not imported.

setup.py

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing.

The `setup.py` file is at the heart of a Python project. It describes all of the metadata about your project. There are quite a few fields you can add to a project to give it a rich set of metadata describing the project. However, there are only three required fields: `name`, `version`, and `packages`. The `name` field must be unique if you wish to publish your package on the Python Package Index (PyPI). The `version` field keeps track of different releases of the project. The `packages` field describes where you've put the Python source code within your project.

This allows you to easily install Python packages. Often it's enough to write:

```
python setup.py install
```

and module will install itself.

Our initial setup.py will also include information about the license and will re-use the README.txt file for the long_description field. This will look like:

```
1 >>> from distutils.core import setup
2 >>> setup(
3 ...     name='pythonCheatsheet',
4 ...     version='0.1',
5 ...     packages=['pipenv',],
6 ...     license='MIT',
7 ...     long_description=open('README.txt').read(),
8 ... )
```

Find more information visit <http://docs.python.org/install/index.html>.

Dataclasses

Dataclasses are python classes but are suited for storing data objects. This module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes.

Features

1. They store data and represent a certain data type. Ex: A number. For people familiar with ORMs, a model instance is a data object. It represents a specific kind of entity. It holds attributes that define or represent the entity.
2. They can be compared to other objects of the same type. Ex: A number can be greater than, less than, or equal to another number.

Python 3.7 provides a decorator `dataclass` that is used to convert a class into a dataclass.

python 2.7

```
1 >>> class Number:
2 ...     def __init__(self, val):
3 ...         self.val = val
4 ...
5 >>> obj = Number(2)
6 >>> obj.val
7 2
```

with dataclass

```
1 >>> @dataclass
2 ... class Number:
3 ...     val: int
4 ...
5 >>> obj = Number(2)
6 >>> obj.val
7 2
```

Default values

It is easy to add default values to the fields of your data class.

```
1 >>> @dataclass
2 ... class Product:
3 ...     name: str
4 ...     count: int = 0
5 ...     price: float = 0.0
6 ...
7 >>> obj = Product("Python")
8 >>> obj.name
9 Python
10 >>> obj.count
11 0
12 >>> obj.price
13 0.0
```

Type hints

It is mandatory to define the data type in dataclass. However, If you don't want specify the datatype then, use `typing.Any`.

```
1 >>> from dataclasses import dataclass
2 >>> from typing import Any
3
4 >>> @dataclass
5 ... class WithoutExplicitTypes:
6 ...     name: Any
7 ...     value: Any = 42
8 ...
```

Virtual Environment

The use of a Virtual Environment is to test python code in encapsulated environments and to also avoid filling the base Python installation with libraries we might use for only one project.

virtualenv

1. Install virtualenv

```
pip install virtualenv
```

2. Install virtualenvwrapper-win (Windows)

```
pip install virtualenvwrapper-win
```

Usage:

1. Make a Virtual Environment

```
mkvirtualenv HelloWold
```

Anything we install now will be specific to this project. And available to the projects we connect to this environment.

2. Set Project Directory

To bind our virtualenv with our current working directory we simply enter:

```
setprojectdir .
```

3. Deactivate

To move onto something else in the command line type 'deactivate' to deactivate your environment.

```
deactivate
```

Notice how the parenthesis disappear.

4. Workon

Open up the command prompt and type 'workon HelloWold' to activate the environment and move into your root project folder

```
workon HelloWold
```

poetry

Poetry is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

1. Install Poetry

```
pip install --user poetry
```

2. Create a new project

```
poetry new my-project
```

This will create a my-project directory:

```
1 my-project
2 └── pyproject.toml
3 └── README.rst
4 └── poetry_demo
5   └── __init__.py
6   └── tests
7     └── __init__.py
8       └── test_poetry_demo.py
```

The pyproject.toml file will orchestrate your project and its dependencies:

```
1 [tool.poetry]
2 name = "my-project"
3 version = "0.1.0"
```

```
4 description = ""
5 authors = ["your name <your@mail.com>"]
6
7 [tool.poetry.dependencies]
8 python = "*"
9
10 [tool.poetry.dev-dependencies]
11 pytest = "^3.4"
```

3. Packages

To add dependencies to your project, you can specify them in the tool.poetry.dependencies section:

```
1 [tool.poetry.dependencies]
2 pendulum = "^1.4"
```

Also, instead of modifying the pyproject.toml file by hand, you can use the add command and it will automatically find a suitable version constraint.

```
$ poetry add pendulum
```

To install the dependencies listed in the pyproject.toml:

```
poetry install
```

To remove dependencies:

```
poetry remove pendulum
```

For more information, check the [documentation](#).

pipenv

Pipenv is a tool that aims to bring the best of all packaging worlds (bundler, composer, npm, cargo, yarn, etc.) to the Python world. Windows is a first-class citizen, in our world.

1. Install pipenv

```
pip install pipenv
```

2. Enter your Project directory and install the Packages for your project

```
1 cd my_project
2 pipenv install <package>
```

Pipenv will install your package and create a Pipfile for you in your project's directory. The Pipfile is used to track which dependencies your project needs in case you need to re-install them.

3. Uninstall Packages

```
pipenv uninstall <package>
```

4. Activate the Virtual Environment associated with your Python project

```
pipenv shell
```

5. Exit the Virtual Environment

```
exit
```

Find more information and a video in docs.pipenv.org.

anaconda

Anaconda is another popular tool to manage python packages.

Where packages, notebooks, projects and environments are shared. Your place for free public conda package hosting.

Usage:

1. Make a Virtual Environment

```
conda create -n HelloWorld
```

2. To use the Virtual Environment, activate it by:

```
conda activate HelloWorld
```

Anything installed now will be specific to the project HelloWorld

3. Exit the Virtual Environment

```
conda deactivate
```

Python Cheatsheet

- About
 - Contribute
 - Read It
 - Python Cheatsheet
 - The Zen of Python
 - Python Basics
 - Math Operators
 - Data Types
 - String Concatenation and Replication
 - Variables
 - Comments
 - The `print()` Function
 - The `input()` Function
 - The `len()` Function
 - The `str(), int(), and float()` Functions
 - Flow Control
 - Comparison Operators
 - Boolean evaluation
 - Boolean Operators
 - Mixing Boolean and Comparison Operators
 - `if` Statements
 - `else` Statements
 - `elif` Statements
 - `while` Loop Statements
 - `break` Statements
 - `continue` Statements
 - `for` Loops and the `range()` Function
 - For else statement
 - Importing Modules
 - Ending a Program Early with `sys.exit()`
 - Functions
 - Return Values and return Statements
 - The `None` Value
 - Keyword Arguments and `print()`
 - Local and Global Scope
 - The `global` Statement
 - Exception Handling
 - Basic exception handling
 - Final code in exception handling
 - Lists
 - Getting Individual Values in a List with Indexes
 - Negative Indexes
 - Getting Sublists with Slices
 - Getting a List's Length with `len()`

- Changing Values in a List with Indexes
- List Concatenation and List Replication
- Removing Values from Lists with del Statements
- Using for Loops with Lists
- Looping Through Multiple Lists with zip()
- The in and not in Operators
- The Multiple Assignment Trick
- Augmented Assignment Operators
- Finding a Value in a List with the index() Method
- Adding Values to Lists with the append() and insert() Methods
- Removing Values from Lists with remove()
- Removing Values from Lists with pop()
- Sorting the Values in a List with the sort() Method
- Tuple Data Type
- Converting Types with the list() and tuple() Functions
- Dictionaries and Structuring Data
 - The keys(), values(), and items() Methods
 - Checking Whether a Key or Value Exists in a Dictionary
 - The get() Method
 - The setdefault() Method
 - Pretty Printing
 - Merge two dictionaries
- sets
 - Initializing a set
 - sets: unordered collections of unique elements
 - set add() and update()
 - set remove() and discard()
 - set union()
 - set intersection
 - set difference
 - set symmetric_difference
- itertools Module
 - accumulate()
 - combinations()
 - combinations_with_replacement()
 - count()
 - cycle()
 - chain()
 - compress()
 - dropwhile()
 - filterfalse()
 - groupby()
 - islice()
 - permutations()
 - product()
 - repeat()
 - starmap()
 - takewhile()
 - tee()
 - zip_longest()
- Comprehensions
 - List comprehension
 - Set comprehension
 - Dict comprehension
- Manipulating Strings
 - Escape Characters

- Raw Strings
- Multiline Strings with Triple Quotes
- Indexing and Slicing Strings
- The in and not in Operators with Strings
- The in and not in Operators with list
- The upper(), lower(), isupper(), and islower() String Methods
- The isX String Methods
- The startswith() and endswith() String Methods
- The join() and split() String Methods
- Justifying Text with rjust(), ljust(), and center()
- Removing Whitespace with strip(), rstrip(), and lstrip()
- Copying and Pasting Strings with the pyperclip Module (need pip install)
- String Formatting
 - % operator
 - String Formatting (str.format)
 - Lazy string formatting
 - Formatted String Literals or f-strings (Python 3.6+)
 - Template Strings
- Regular Expressions
 - Matching Regex Objects
 - Grouping with Parentheses
 - Matching Multiple Groups with the Pipe
 - Optional Matching with the Question Mark
 - Matching Zero or More with the Star
 - Matching One or More with the Plus
 - Matching Specific Repetitions with Curly Brackets
 - Greedy and Nongreedy Matching
 - The findall() Method
 - Making Your Own Character Classes
 - The Caret and Dollar Sign Characters
 - The Wildcard Character
 - Matching Everything with Dot-Star
 - Matching Newlines with the Dot Character
 - Review of Regex Symbols
 - Case-Insensitive Matching
 - Substituting Strings with the sub() Method
 - Managing Complex Regexes
- Handling File and Directory Paths
 - Backslash on Windows and Forward Slash on OS X and Linux
 - The Current Working Directory
 - Creating New Folders
 - Absolute vs. Relative Paths
 - Handling Absolute and Relative Paths
 - Checking Path Validity
 - Finding File Sizes and Folder Contents
 - Copying Files and Folders
 - Moving and Renaming Files and Folders
 - Permanently Deleting Files and Folders
 - Safe Deletes with the send2trash Module
 - Walking a Directory Tree
- Reading and Writing Files
 - The File Reading/Writing Process
 - Opening and reading files with the open() function
 - Writing to Files
 - Saving Variables with the shelve Module
 - Saving Variables with the pprint.pformat() Function

- [Reading ZIP Files](#)
- [Extracting from ZIP Files](#)
- [Creating and Adding to ZIP Files](#)
- [JSON, YAML and configuration files](#)
 - [JSON](#)
 - [YAML](#)
 - [Anyconfig](#)
- [Debugging](#)
 - [Raising Exceptions](#)
 - [Getting the Traceback as a String](#)
 - [Assertions](#)
 - [Logging](#)
 - [Logging Levels](#)
 - [Disabling Logging](#)
 - [Logging to a File](#)
- [Lambda Functions](#)
- [Ternary Conditional Operator](#)
- [args and kwargs](#)
 - [Things to Remember\(args\)](#)
 - [Things to Remember\(kwargs\)](#)
- [Context Manager](#)
 - [with statement](#)
 - [Writing your own contextmanager using generator syntax](#)
- [__main__ Top-level script environment](#)
 - [Advantages](#)
- [setup.py](#)
- [Dataclasses](#)
 - [Features](#)
 - [Default values](#)
 - [Type hints](#)
- [Virtual Environment](#)
 - [virtualenv](#)
 - [poetry](#)
 - [pipenv](#)
 - [anaconda](#)

The Zen of Python

From the PEP 20 – The Zen of Python:

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

```

1  >>> import this
2  The Zen of Python, by Tim Peters
3
4  Beautiful is better than ugly.
5  Explicit is better than implicit.
6  Simple is better than complex.
7  Complex is better than complicated.
8  Flat is better than nested.
9  Sparse is better than dense.
10 Readability counts.
11 Special cases aren't special enough to break the rules.
12 Although practicality beats purity.
13 Errors should never pass silently.
14 Unless explicitly silenced.
15 In the face of ambiguity, refuse the temptation to guess.
16 There should be one-- and preferably only one --obvious way to do it.

```

```
17 Although that way may not be obvious at first unless you're Dutch.  
18 Now is better than never.  
19 Although never is often better than *right* now.  
20 If the implementation is hard to explain, it's a bad idea.  
21 If the implementation is easy to explain, it may be a good idea.  
22 Namespaces are one honking great idea -- let's do more of those!
```

[Return to the Top](#)

Python Basics

Math Operators

From **Highest** to **Lowest** precedence:

Operators	Operation	Example
**	Exponent	2 ** 3 = 8
%	Modulus/Remainder	22 % 8 = 6
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 9
-	Subtraction	5 - 2 = 3
+	Addition	2 + 2 = 4

Examples of expressions in the interactive shell:

```
1 >>> 2 + 3 * 6  
2 20
```

```
1 >>> (2 + 3) * 6  
2 30
```

```
1 >>> 2 ** 8  
2 256
```

```
1 >>> 23 // 7  
2 3
```

```
1 >>> 23 % 7  
2 2
```

```
1 >>> (5 - 1) * ((7 + 1) / (3 - 1))
```

```
2 16.0
```

[Return to the Top](#)

Data Types

Data Type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

[Return to the Top](#)

String Concatenation and Replication

String concatenation:

```
1 >>> 'Alice' 'Bob'  
2 'AliceBob'
```

Note: Avoid `+` operator for string concatenation. Prefer string formatting.

String Replication:

```
1 >>> 'Alice' * 5  
2 'AliceAliceAliceAliceAlice'
```

[Return to the Top](#)

Variables

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (`_`) character.
3. It can't begin with a number.
4. Variable name starting with an underscore (`_`) are considered as "unuseful".

Example:

```
1 >>> spam = 'Hello'  
2 >>> spam  
3 'Hello'
```

```
>>> _spam = 'Hello'
```

```
_spam should not be used again in the code.
```

[Return to the Top](#)

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

```
1 # This is a
2 # multiline comment
```

Code with a comment:

```
a = 1 # initialization
```

Please note the two spaces in front of the comment.

Function docstring:

```
1 def foo():
2     """
3     This is a function docstring
4     You can also use:
5     ''' Function Docstring '''
6     """
```

[Return to the Top](#)

The print() Function

```
1 >>> print('Hello world!')
2 Hello world!
```

```
1 >>> a = 1
2 >>> print('Hello world!', a)
3 Hello world! 1
```

[Return to the Top](#)

The input() Function

Example Code:

```
1 >>> print('What is your name?')    # ask for their name
2 >>> myName = input()
3 >>> print('It is good to meet you, {}'.format(myName))
4 What is your name?
5 Al
6 It is good to meet you, Al
```

[Return to the Top](#)

The `len()` Function

Evaluates to the integer value of the number of characters in a string:

```
1 >>> len('hello')
2 5
```

Note: test of emptiness of strings, lists, dictionary, etc, should **not** use `len`, but prefer direct boolean evaluation.

```
1 >>> a = [1, 2, 3]
2 >>> if a:
3 >>>     print("the list is not empty!")
```

[Return to the Top](#)

The `str()`, `int()`, and `float()` Functions

Integer to String or Float:

```
1 >>> str(29)
2 '29'
```

```
1 >>> print('I am {} years old.'.format(str(29)))
2 I am 29 years old.
```

```
1 >>> str(-3.14)
2 '-3.14'
```

Float to Integer:

```
1 >>> int(7.7)
2 7
```

```
1 >>> int(7.7) + 1
2 8
```

[Return to the Top](#)

Flow Control

Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater Than
<code><=</code>	Less than or Equal to
<code>>=</code>	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
1 >>> 42 == 42
2 True
```

```
1 >>> 40 == 42
2 False
```

```
1 >>> 'hello' == 'hello'
2 True
```

```
1 >>> 'hello' == 'Hello'
2 False
```

```
1 >>> 'dog' != 'cat'
2 True
```

```
1 >>> 42 == 42.0
2 True
```

```
1 >>> 42 == '42'
2 False
```

Boolean evaluation

Never use `==` or `!=` operator to evaluate boolean operation. Use the `is` or `is not` operators, or use implicit boolean evaluation.

NO (even if they are valid Python):

```
1 >>> True == True
2 True
```

```
1 >>> True != False
2 True
```

YES (even if they are valid Python):

```
1 >>> True is True
2 True
```

```
1 >>> True is not False
2 True
```

These statements are equivalent:

```
1 >>> if a is True:
2 >>>     pass
3 >>> if a is not False:
4 >>>     pass
5 >>> if a:
6 >>>     pass
```

And these as well:

```
1 >>> if a is False:
2 >>>     pass
3 >>> if a is not True:
4 >>>     pass
5 >>> if not a:
6 >>>     pass
```

[Return to the Top](#)

Boolean Operators

There are three Boolean operators: and, or, and not.

The *and* Operator's *Truth Table*:

Expression

Evaluates to

True and True	True
True and False	False
False and True	False
False and False	False

The *or* Operator's *Truth Table*:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The *not* Operator's *Truth Table*:

Expression	Evaluates to
not True	False
not False	True

[Return to the Top](#)

Mixing Boolean and Comparison Operators

```
1 >>> (4 < 5) and (5 < 6)
2 True
```

```
1 >>> (4 < 5) and (9 < 6)
2 False
```

```
1 >>> (1 == 2) or (2 == 2)
2 True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
1 >>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
2 True
```

[Return to the Top](#)

if Statements

```
1 if name == 'Alice':  
2     print('Hi, Alice.')
```

[Return to the Top](#)

else Statements

```
1 name = 'Bob'  
2 if name == 'Alice':  
3     print('Hi, Alice.')4 else:  
5     print('Hello, stranger.')
```

[Return to the Top](#)

elif Statements

```
1 name = 'Bob'  
2 age = 5  
3 if name == 'Alice':  
4     print('Hi, Alice.')5 elif age < 12:  
6     print('You are not Alice, kiddo.')
```

```
1 name = 'Bob'  
2 age = 30  
3 if name == 'Alice':  
4     print('Hi, Alice.')5 elif age < 12:  
6     print('You are not Alice, kiddo.')7 else:  
8     print('You are neither Alice nor a little kid.')
```

[Return to the Top](#)

while Loop Statements

```
1 spam = 0  
2 while spam < 5:  
3     print('Hello, world.')4     spam = spam + 1
```

[Return to the Top](#)

break Statements

If the execution reaches a break statement, it immediately exits the while loop's clause:

```
1 while True:  
2     print('Please type your name.')3     name = input()  
4     if name == 'your name':
```

```
5     break
6 print('Thank you!')
```

[Return to the Top](#)

continue Statements

When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop.

```
1 while True:
2     print('Who are you?')
3     name = input()
4     if name != 'Joe':
5         continue
6     print('Hello, Joe. What is the password? (It is a fish.)')
7     password = input()
8     if password == 'swordfish':
9         break
10    print('Access granted.')
```

[Return to the Top](#)

for Loops and the range() Function

```
1 >>> print('My name is')
2 >>> for i in range(5):
3 >>>     print('Jimmy Five Times {}'.format(str(i)))
4 My name is
5 Jimmy Five Times (0)
6 Jimmy Five Times (1)
7 Jimmy Five Times (2)
8 Jimmy Five Times (3)
9 Jimmy Five Times (4)
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
1 >>> for i in range(0, 10, 2):
2 >>>     print(i)
3 0
4 2
5 4
6 6
7 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
1 >>> for i in range(5, -1, -1):
2 >>>     print(i)
3 5
4 4
5 3
6 2
7 1
8 0
```

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
1 >>> for i in [1, 2, 3, 4, 5]:  
2 >>>     if i == 3:  
3 >>>         break  
4 >>> else:  
5 >>>     print("only executed when no item of the list is equal to 3")
```

[Return to the Top](#)

Importing Modules

```
1 import random  
2 for i in range(5):  
3     print(random.randint(1, 10))
```

```
import random, sys, os, math
```

```
from random import *
```

[Return to the Top](#)

Ending a Program Early with `sys.exit()`

```
1 import sys  
2  
3 while True:  
4     print('Type exit to exit.')  
5     response = input()  
6     if response == 'exit':  
7         sys.exit()  
8     print('You typed {}'.format(response))
```

[Return to the Top](#)

Functions

```
1 >>> def hello(name):  
2 >>>     print('Hello {}'.format(name))  
3 >>>  
4 >>> hello('Alice')  
5 >>> hello('Bob')  
6 Hello Alice  
7 Hello Bob
```

[Return to the Top](#)

Return Values and return Statements

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

- The `return` keyword.
- The value or expression that the function should return.

```
1 import random
2 def getAnswer(answerNumber):
3     if answerNumber == 1:
4         return 'It is certain'
5     elif answerNumber == 2:
6         return 'It is decidedly so'
7     elif answerNumber == 3:
8         return 'Yes'
9     elif answerNumber == 4:
10        return 'Reply hazy try again'
11    elif answerNumber == 5:
12        return 'Ask again later'
13    elif answerNumber == 6:
14        return 'Concentrate and ask again'
15    elif answerNumber == 7:
16        return 'My reply is no'
17    elif answerNumber == 8:
18        return 'Outlook not so good'
19    elif answerNumber == 9:
20        return 'Very doubtful'
21
22 r = random.randint(1, 9)
23 fortune = getAnswer(r)
24 print(fortune)
```

[Return to the Top](#)

The None Value

```
1 >>> spam = print('Hello!')
2 Hello!
```

```
1 >>> spam is None
2 True
```

Note: never compare to `None` with the `==` operator. Always use `is`.

[Return to the Top](#)

Keyword Arguments and `print()`

```
1 >>> print('Hello', end='')
2 >>> print('World')
3 HelloWorld
```

```
1 >>> print('cats', 'dogs', 'mice')
2 cats dogs mice
```

```
1 >>> print('cats', 'dogs', 'mice', sep=',')
2 cats,dogs,mice
```

[Return to the Top](#)

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

[Return to the Top](#)

The global Statement

If you need to modify a global variable from within a function, use the global statement:

```
1 >>> def spam():
2 >>>     global eggs
3 >>>     eggs = 'spam'
4 >>>
5 >>> eggs = 'global'
6 >>> spam()
7 >>> print(eggs)
8 spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

[Return to the Top](#)

Exception Handling

Basic exception handling

```
1 >>> def spam(divideBy):
2 >>>     try:
3 >>>         return 42 / divideBy
4 >>>     except ZeroDivisionError as e:
5 >>>         print('Error: Invalid argument: {}'.format(e))
6 >>>
7 >>> print(spam(2))
8 >>> print(spam(12))
9 >>> print(spam(0))
10 >>> print(spam(1))
11 21.0
12 3.5
13 Error: Invalid argument: division by zero
14 None
15 42.0
```

[Return to the Top](#)

Final code in exception handling

Code inside the `finally` section is always executed, no matter if an exception has been raised or not, and even if an exception is not caught.

```
1 >>> def spam(divideBy):
2     try:
3         return 42 / divideBy
4     except ZeroDivisionError as e:
5         print('Error: Invalid argument: {}'.format(e))
6     finally:
7         print("-- division finished --")
8     print(spam(2))
9 -- division finished --
10 21.0
11 >>> print(spam(12))
12 -- division finished --
13 3.5
14 >>> print(spam(0))
15 Error: Invalid Argument division by zero
16 -- division finished --
17 None
18 >>> print(spam(1))
19 -- division finished --
20 42.0
```

[Return to the Top](#)

Lists

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2
3 >>> spam
4 ['cat', 'bat', 'rat', 'elephant']
```

[Return to the Top](#)

Getting Individual Values in a List with Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[0]
3 'cat'
```

```
1 >>> spam[1]
2 'bat'
```

```
1 >>> spam[2]
2 'rat'
```

```
1 >>> spam[3]
2 'elephant'
```

[Return to the Top](#)

Negative Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[-1]
3 'elephant'
```

```
1 >>> spam[-3]
2 'bat'
```

```
1 >>> 'The {} is afraid of the {}.'.format(spam[-1], spam[-3])
2 'The elephant is afraid of the bat.'
```

[Return to the Top](#)

Getting Sublists with Slices

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[0:4]
3 ['cat', 'bat', 'rat', 'elephant']
```

```
1 >>> spam[1:3]
2 ['bat', 'rat']
```

```
1 >>> spam[0:-1]
2 ['cat', 'bat', 'rat']
```

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[:2]
3 ['cat', 'bat']
```

```
1 >>> spam[1:]
2 ['bat', 'rat', 'elephant']
```

Slicing the complete list will perform a copy:

```
1 >>> spam2 = spam[:]
2 ['cat', 'bat', 'rat', 'elephant']
3 >>> spam.append('dog')
```

```
4 >>> spam
5 ['cat', 'bat', 'rat', 'elephant', 'dog']
6 >>> spam2
7 ['cat', 'bat', 'rat', 'elephant']
```

[Return to the Top](#)

Getting a List's Length with len()

```
1 >>> spam = ['cat', 'dog', 'moose']
2 >>> len(spam)
3 3
```

[Return to the Top](#)

Changing Values in a List with Indexes

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> spam[1] = 'aardvark'
3
4 >>> spam
5 ['cat', 'aardvark', 'rat', 'elephant']
6
7 >>> spam[2] = spam[1]
8
9 >>> spam
10 ['cat', 'aardvark', 'aardvark', 'elephant']
11
12 >>> spam[-1] = 12345
13
14 >>> spam
15 ['cat', 'aardvark', 'aardvark', 12345]
```

[Return to the Top](#)

List Concatenation and List Replication

```
1 >>> [1, 2, 3] + ['A', 'B', 'C']
2 [1, 2, 3, 'A', 'B', 'C']
3
4 >>> ['X', 'Y', 'Z'] * 3
5 ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
6
7 >>> spam = [1, 2, 3]
8
9 >>> spam = spam + ['A', 'B', 'C']
10
11 >>> spam
12 [1, 2, 3, 'A', 'B', 'C']
```

[Return to the Top](#)

Removing Values from Lists with del Statements

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2 >>> del spam[2]
3 >>> spam
```

```
4 ['cat', 'bat', 'elephant']
```

```
1 >>> del spam[2]
2 >>> spam
3 ['cat', 'bat']
```

[Return to the Top](#)

Using for Loops with Lists

```
1 >>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
2 >>> for i, supply in enumerate(supplies):
3 >>>     print('Index {} in supplies is: {}'.format(str(i), supply))
4 Index 0 in supplies is: pens
5 Index 1 in supplies is: staplers
6 Index 2 in supplies is: flame-throwers
7 Index 3 in supplies is: binders
```

[Return to the Top](#)

Looping Through Multiple Lists with zip()

```
1 >>> name = ['Pete', 'John', 'Elizabeth']
2 >>> age = [6, 23, 44]
3 >>> for n, a in zip(name, age):
4 >>>     print('{} is {} years old'.format(n, a))
5 Pete is 6 years old
6 John is 23 years old
7 Elizabeth is 44 years old
```

The in and not in Operators

```
1 >>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
2 True
```

```
1 >>> spam = ['hello', 'hi', 'howdy', 'heyas']
2 >>> 'cat' in spam
3 False
```

```
1 >>> 'howdy' not in spam
2 False
```

```
1 >>> 'cat' not in spam
2 True
```

[Return to the Top](#)

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
1 >>> cat = ['fat', 'orange', 'loud']
2
3 >>> size = cat[0]
4
5 >>> color = cat[1]
6
7 >>> disposition = cat[2]
```

You could type this line of code:

```
1 >>> cat = ['fat', 'orange', 'loud']
2
3 >>> size, color, disposition = cat
```

The multiple assignment trick can also be used to swap the values in two variables:

```
1 >>> a, b = 'Alice', 'Bob'
2 >>> a, b = b, a
3 >>> print(a)
4 'Bob'
```

```
1 >>> print(b)
2 'Alice'
```

[Return to the Top](#)

Augmented Assignment Operators

Operator	Equivalent
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1
spam %= 1	spam = spam % 1

Examples:

```
1 >>> spam = 'Hello'
2 >>> spam += ' world!'
3 >>> spam
4 'Hello world!'
5
```

```
6 >>> bacon = ['Zophie']
7 >>> bacon *= 3
8 >>> bacon
9 ['Zophie', 'Zophie', 'Zophie']
```

[Return to the Top](#)

Finding a Value in a List with the `index()` Method

```
1 >>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
2
3 >>> spam.index('Pooka')
4 1
```

[Return to the Top](#)

Adding Values to Lists with the `append()` and `insert()` Methods

append():

```
1 >>> spam = ['cat', 'dog', 'bat']
2
3 >>> spam.append('moose')
4
5 >>> spam
6 ['cat', 'dog', 'bat', 'moose']
```

insert():

```
1 >>> spam = ['cat', 'dog', 'bat']
2
3 >>> spam.insert(1, 'chicken')
4
5 >>> spam
6 ['cat', 'chicken', 'dog', 'bat']
```

[Return to the Top](#)

Removing Values from Lists with `remove()`

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2
3 >>> spam.remove('bat')
4
5 >>> spam
6 ['cat', 'rat', 'elephant']
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

[Return to the Top](#)

Removing Values from Lists with `pop()`

```
1 >>> spam = ['cat', 'bat', 'rat', 'elephant']
2
3 >>> spam.pop()
4 'elephant'
5
6 >>> spam
7 ['cat', 'bat', 'rat']
8
9 >>> spam.pop(0)
10 'cat'
11
12 >>> spam
13 ['bat', 'rat']
```

[Return to the Top](#)

Sorting the Values in a List with the sort() Method

```
1 >>> spam = [2, 5, 3.14, 1, -7]
2 >>> spam.sort()
3 >>> spam
4 [-7, 1, 2, 3.14, 5]
```

```
1 >>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
2 >>> spam.sort()
3 >>> spam
4 ['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order:

```
1 >>> spam.sort(reverse=True)
2 >>> spam
3 ['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

If you need to sort the values in regular alphabetical order, pass str.lower for the key keyword argument in the sort() method call:

```
1 >>> spam = ['a', 'z', 'A', 'Z']
2 >>> spam.sort(key=str.lower)
3 >>> spam
4 ['a', 'A', 'z', 'Z']
```

You can use the built-in function `sorted` to return a new list:

```
1 >>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
2 >>> sorted(spam)
3 ['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

[Return to the Top](#)

Tuple Data Type

```
1 >>> eggs = ('hello', 42, 0.5)
2 >>> eggs[0]
3 'hello'
```

```
1 >>> eggs[1:3]
2 (42, 0.5)
```

```
1 >>> len(eggs)
2 3
```

The main way that tuples are different from lists is that tuples, like strings, are immutable.

[Return to the Top](#)

Converting Types with the `list()` and `tuple()` Functions

```
1 >>> tuple(['cat', 'dog', 5])
2 ('cat', 'dog', 5)
```

```
1 >>> list(('cat', 'dog', 5))
2 ['cat', 'dog', 5]
```

```
1 >>> list('hello')
2 ['h', 'e', 'l', 'l', 'o']
```

[Return to the Top](#)

Dictionaries and Structuring Data

Example Dictionary:

```
myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

[Return to the Top](#)

The `keys()`, `values()`, and `items()` Methods

`values()`:

```
1 >>> spam = {'color': 'red', 'age': 42}
2 >>> for v in spam.values():
3     print(v)
4 red
5 42
```

keys():

```
1 >>> for k in spam.keys():
2 >>>     print(k)
3 color
4 age
```

items():

```
1 >>> for i in spam.items():
2 >>>     print(i)
3 ('color', 'red')
4 ('age', 42)
```

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.

```
1 >>> spam = {'color': 'red', 'age': 42}
2 >>>
3 >>> for k, v in spam.items():
4 >>>     print('Key: {} Value: {}'.format(k, str(v)))
5 Key: age Value: 42
6 Key: color Value: red
```

[Return to the Top](#)

Checking Whether a Key or Value Exists in a Dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
1 >>> 'name' in spam.keys()
2 True
```

```
1 >>> 'Zophie' in spam.values()
2 True
```

```
1 >>> # You can omit the call to keys() when checking for a key
2 >>> 'color' in spam
3 False
```

```
1 >>> 'color' not in spam
2 True
```

[Return to the Top](#)

The get() Method

Get has two parameters: key and default value if the key did not exist

```
1 >>> picnic_items = {'apples': 5, 'cups': 2}
2
3 >>> 'I am bringing {} cups.'.format(str(picnic_items.get('cups', 0)))
4 'I am bringing 2 cups.'
```

```
1 >>> 'I am bringing {} eggs.'.format(str(picnic_items.get('eggs', 0)))
2 'I am bringing 0 eggs.'
```

[Return to the Top](#)

The `setdefault()` Method

Let's consider this code:

```
1 spam = {'name': 'Pooka', 'age': 5}
2
3 if 'color' not in spam:
4     spam['color'] = 'black'
```

Using `setdefault` we could write the same code more succinctly:

```
1 >>> spam = {'name': 'Pooka', 'age': 5}
2 >>> spam.setdefault('color', 'black')
3 'black'
```

```
1 >>> spam
2 {'color': 'black', 'age': 5, 'name': 'Pooka'}
```

```
1 >>> spam.setdefault('color', 'white')
2 'black'
```

```
1 >>> spam
2 {'color': 'black', 'age': 5, 'name': 'Pooka'}
```

[Return to the Top](#)

Pretty Printing

```
1 >>> import pprint
2 >>>
3 >>> message = 'It was a bright cold day in April, and the clocks were striking
4 >>> thirteen.'
5 >>> count = {}
6 >>>
7 >>> for character in message:
```

```
8 >>>     count.setdefault(character, 0)
9 >>>     count[character] = count[character] + 1
10 >>>
11 >>> pprint.pprint(count)
12 {' ': 13,
13 ',': 1,
14 '.': 1,
15 'A': 1,
16 'I': 1,
17 'a': 4,
18 'b': 1,
19 'c': 3,
20 'd': 3,
21 'e': 5,
22 'g': 2,
23 'h': 3,
24 'i': 6,
25 'k': 2,
26 'l': 3,
27 'n': 4,
28 'o': 2,
29 'p': 1,
30 'r': 5,
31 's': 3,
32 't': 6,
33 'w': 2,
34 'y': 1}
```

[Return to the Top](#)

Merge two dictionaries

```
1 # in Python 3.5+:
2 >>> x = {'a': 1, 'b': 2}
3 >>> y = {'b': 3, 'c': 4}
4 >>> z = {**x, **y}
5 >>> z
6 {'c': 4, 'a': 1, 'b': 3}
7
8 # in Python 2.7
9 >>> z = dict(x, **y)
10 >>> z
11 {'c': 4, 'a': 1, 'b': 3}
```

sets

From the Python 3 [documentation](#)

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.

Initializing a set

There are two ways to create sets: using curly braces `{}` and the built-in function `set()`

```
1 >>> s = {1, 2, 3}
2 >>> s = set([1, 2, 3])
```

When creating an empty set, be sure to not use the curly braces `{}` or you will get an empty dictionary instead.

```
1 >>> s = []
2 >>> type(s)
3 <class 'dict'>
```

sets: unordered collections of unique elements

A set automatically remove all the duplicate values.

```
1 >>> s = {1, 2, 3, 2, 3, 4}
2 >>> s
3 {1, 2, 3, 4}
```

And as an unordered data type, they can't be indexed.

```
1 >>> s = {1, 2, 3}
2 >>> s[0]
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 TypeError: 'set' object does not support indexing
6 >>>
```

set add() and update()

Using the `add()` method we can add a single element to the set.

```
1 >>> s = {1, 2, 3}
2 >>> s.add(4)
3 >>> s
4 {1, 2, 3, 4}
```

And with `update()`, multiple ones .

```
1 >>> s = {1, 2, 3}
2 >>> s.update([2, 3, 4, 5, 6])
3 >>> s
4 {1, 2, 3, 4, 5, 6} # remember, sets automatically remove duplicates
```

set remove() and discard()

Both methods will remove an element from the set, but `remove()` will raise a `key error` if the value doesn't exist.

```
1 >>> s = {1, 2, 3}
2 >>> s.remove(3)
3 >>> s
4 {1, 2}
5 >>> s.remove(3)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 KeyError: 3
```

```
discard() won't raise any errors.
```

```
1 >>> s = {1, 2, 3}
2 >>> s.discard(3)
3 >>> s
4 {1, 2}
5 >>> s.discard(3)
6 >>>
```

set union()

`union()` or `|` will create a new set that contains all the elements from the sets provided.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {3, 4, 5}
3 >>> s1.union(s2) # or 's1 | s2'
4 {1, 2, 3, 4, 5}
```

set intersection

`intersection` or `&` will return a set containing only the elements that are common to all of them.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s3 = {3, 4, 5}
4 >>> s1.intersection(s2, s3) # or 's1 & s2 & s3'
5 {3}
```

set difference

`difference` or `-` will return only the elements that are unique to the first set (invoked set).

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s1.difference(s2) # or 's1 - s2'
4 {1}
5 >>> s2.difference(s1) # or 's2 - s1'
6 {4}
```

set symmetric_difference

`symmetric_difference` or `^` will return all the elements that are not common between them.

```
1 >>> s1 = {1, 2, 3}
2 >>> s2 = {2, 3, 4}
3 >>> s1.symmetric_difference(s2) # or 's1 ^ s2'
4 {1, 4}
```

[Return to the Top](#)

itertools Module

The *itertools* module is a collection of tools intended to be fast and use memory efficiently when handling iterators (like [lists](#) or [dictionaries](#)).

From the official [Python 3.x documentation](#):

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

The *itertools* module comes in the standard library and must be imported.

The [operator](#) module will also be used. This module is not necessary when using itertools, but needed for some of the examples below.

Return to the Top

accumulate()

Makes an iterator that returns the results of a function.

```
itertools.accumulate(iterable[, func])
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5]
2 >>> result = itertools.accumulate(data, operator.mul)
3 >>> for each in result:
4 >>>     print(each)
5 1
6 2
7 6
8 24
9 120
```

The `operator.mul` takes two numbers and multiplies them:

```
1 operator.mul(1, 2)
2 2
3 operator.mul(2, 3)
4 6
5 operator.mul(6, 4)
6 24
7 operator.mul(24, 5)
8 120
```

Passing a function is optional:

```
1 >>> data = [5, 2, 6, 4, 5, 9, 1]
2 >>> result = itertools.accumulate(data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 7
7 13
8 17
9 22
10 31
11 32
```

If no function is designated the items will be summed:

```
1  5
2  5 + 2 = 7
3  7 + 6 = 13
4  13 + 4 = 17
5  17 + 5 = 22
6  22 + 9 = 31
7  31 + 1 = 32
```

[Return to the Top](#)

combinations()

Takes an iterable and a integer. This will create all the unique combination that have r members.

```
itertools.combinations(iterable, r)
```

Example:

```
1  >>> shapes = ['circle', 'triangle', 'square',]
2  >>> result = itertools.combinations(shapes, 2)
3  >>> for each in result:
4  >>>     print(each)
5  ('circle', 'triangle')
6  ('circle', 'square')
7  ('triangle', 'square')
```

[Return to the Top](#)

combinations_with_replacement()

Just like combinations(), but allows individual elements to be repeated more than once.

```
itertools.combinations_with_replacement(iterable, r)
```

Example:

```
1  >>> shapes = ['circle', 'triangle', 'square']
2  >>> result = itertools.combinations_with_replacement(shapes, 2)
3  >>> for each in result:
4  >>>     print(each)
5  ('circle', 'circle')
6  ('circle', 'triangle')
7  ('circle', 'square')
8  ('triangle', 'triangle')
9  ('triangle', 'square')
10 ('square', 'square')
```

[Return to the Top](#)

count()

Makes an iterator that returns evenly spaced values starting with number start.

```
itertools.count(start=0, step=1)
```

Example:

```
1 >>> for i in itertools.count(10,3):
2 >>>     print(i)
3 >>>     if i > 20:
4 >>>         break
5 10
6 13
7 16
8 19
9 22
```

[Return to the Top](#)

cycle()

This function cycles through an iterator endlessly.

```
itertools.cycle(iterable)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet']
2 >>> for color in itertools.cycle(colors):
3 >>>     print(color)
4 red
5 orange
6 yellow
7 green
8 blue
9 violet
10 red
11 orange
```

When reached the end of the iterable it start over again from the beginning.

[Return to the Top](#)

chain()

Take a series of iterables and return them as one long iterable.

```
itertools.chain(*iterables)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> shapes = ['circle', 'triangle', 'square', 'pentagon']
3 >>> result = itertools.chain(colors, shapes)
4 >>> for each in result:
5 >>>     print(each)
6 red
7 orange
8 yellow
9 green
10 blue
11 circle
12 triangle
13 square
14 pentagon
```

[Return to the Top](#)

compress()

Filters one iterable with another.

```
itertools.compress(data, selectors)
```

Example:

```
1 >>> shapes = ['circle', 'triangle', 'square', 'pentagon']
2 >>> selections = [True, False, True, False]
3 >>> result = itertools.compress(shapes, selections)
4 >>> for each in result:
5 >>>     print(each)
6 circle
7 square
```

[Return to the Top](#)

dropwhile()

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.

```
itertools.dropwhile(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.dropwhile(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 6
7 7
8 8
9 9
10 10
11 1
```

[Return to the Top](#)

filterfalse()

Makes an iterator that filters elements from iterable returning only those for which the predicate is False.

```
itertools.filterfalse(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.filterfalse(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 5
6 6
7 7
8 8
9 9
10 10
```

[Return to the Top](#)

groupby()

Simply put, this function groups things together.

```
itertools.groupby(iterable, key=None)
```

Example:

```
1 >>> robots = [{ 
2     'name': 'blaster',
3     'faction': 'autobot'
4 }, { 
5     'name': 'galvatron',
6     'faction': 'decepticon'
7 }, { 
8     'name': 'jazz',
9     'faction': 'autobot'
10 }, { 
11     'name': 'metroplex',
12     'faction': 'autobot'
13 }, { 
14     'name': 'megatron',
15     'faction': 'decepticon'
16 }, { 
17     'name': 'starcream',
18     'faction': 'decepticon'
19 }]
20 >>> for key, group in itertools.groupby(robots, key=lambda x: x['faction']):
21 >>>     print(key)
22 >>>     print(list(group))
23 autobot
24 [{"name": "blaster", "faction": "autobot"}]
25 decepticon
26 [{"name": "galvatron", "faction": "decepticon"}]
27 autobot
28 [{"name": "jazz", "faction": "autobot"}, {"name": "metroplex", "faction": "autobot"}]
```

```
29 decepticon
30 [{"name": "megatron", "faction": "decepticon"}, {"name": "starcream", "faction": "decepticon"}]
```

[Return to the Top](#)

islice()

This function is very much like slices. This allows you to cut out a piece of an iterable.

```
itertools.islice(iterable, start, stop[, step])
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]
2 >>> few_colors = itertools.islice(colors, 2)
3 >>> for each in few_colors:
4 >>>     print(each)
5 red
6 orange
```

[Return to the Top](#)

permutations()

```
itertools.permutations(iterable, r=None)
```

Example:

```
1 >>> alpha_data = ['a', 'b', 'c']
2 >>> result = itertools.permutations(alpha_data)
3 >>> for each in result:
4 >>>     print(each)
5 ('a', 'b', 'c')
6 ('a', 'c', 'b')
7 ('b', 'a', 'c')
8 ('b', 'c', 'a')
9 ('c', 'a', 'b')
10 ('c', 'b', 'a')
```

[Return to the Top](#)

product()

Creates the cartesian products from a series of iterables.

```
1 >>> num_data = [1, 2, 3]
2 >>> alpha_data = ['a', 'b', 'c']
3 >>> result = itertools.product(num_data, alpha_data)
4 >>> for each in result:
5     print(each)
6 (1, 'a')
7 (1, 'b')
```

```
8 (1, 'c')
9 (2, 'a')
10 (2, 'b')
11 (2, 'c')
12 (3, 'a')
13 (3, 'b')
14 (3, 'c')
```

[Return to the Top](#)

repeat()

This function will repeat an object over and over again. Unless, there is a times argument.

```
itertools.repeat(object[, times])
```

Example:

```
1 >>> for i in itertools.repeat("spam", 3):
2     print(i)
3 spam
4 spam
5 spam
```

[Return to the Top](#)

starmap()

Makes an iterator that computes the function using arguments obtained from the iterable.

```
itertools.starmap(function, iterable)
```

Example:

```
1 >>> data = [(2, 6), (8, 4), (7, 3)]
2 >>> result = itertools.starmap(operator.mul, data)
3 >>> for each in result:
4     print(each)
5 12
6 32
7 21
```

[Return to the Top](#)

takewhile()

The opposite of dropwhile(). Makes an iterator and returns elements from the iterable as long as the predicate is true.

```
itertools.takewhile(predicate, iterable)
```

Example:

```
1 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
2 >>> result = itertools.takewhile(lambda x: x<5, data)
3 >>> for each in result:
4 >>>     print(each)
5 1
6 2
7 3
8 4
```

[Return to the Top](#)

tee()

Return n independent iterators from a single iterable.

```
itertools.tee(iterable, n=2)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> alpha_colors, beta_colors = itertools.tee(colors)
3 >>> for each in alpha_colors:
4 >>>     print(each)
5 red
6 orange
7 yellow
8 green
9 blue
```

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
2 >>> alpha_colors, beta_colors = itertools.tee(colors)
3 >>> for each in beta_colors:
4 >>>     print(each)
5 red
6 orange
7 yellow
8 green
9 blue
```

[Return to the Top](#)

zip_longest()

Makes an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

```
itertools.zip_longest(*iterables, fillvalue=None)
```

Example:

```
1 >>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]
2 >>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,]
3 >>> for each in itertools.zip_longest(colors, data, fillvalue=None):
4 >>>     print(each)
5 ('red', 1)
6 ('orange', 2)
7 ('yellow', 3)
8 ('green', 4)
9 ('blue', 5)
10 (None, 6)
11 (None, 7)
12 (None, 8)
13 (None, 9)
14 (None, 10)
```

[Return to the Top](#)

Comprehensions

List comprehension

```
1 >>> a = [1, 3, 5, 7, 9, 11]
2
3 >>> [i - 1 for i in a]
4 [0, 2, 4, 6, 8, 10]
```

Set comprehension

```
1 >>> b = {"abc", "def"}
2 >>> {s.upper() for s in b}
3 {"ABC", "DEF"}
```

Dict comprehension

```
1 >>> c = {'name': 'Pooka', 'age': 5}
2 >>> {v: k for k, v in c.items()}
3 {'Pooka': 'name', 5: 'age'}
```

A List comprehension can be generated from a dictionary:

```
1 >>> c = {'name': 'Pooka', 'first_name': 'Oooka'}
2 >>> [ "{}:{}".format(k.upper(), v.upper()) for k, v in c.items()]
3 ['NAME:POOKA', 'FIRST_NAME:OOOKA']
```

Manipulating Strings

Escape Characters

Escape character

Prints as

\'

Single quote

\"	Double quote
\t	Tab
\n	Newline (line break)
\\"	Backslash

Example:

```

1 >>> print("Hello there!\nHow are you?\nI'm doing fine.")
2 Hello there!
3 How are you?
4 I'm doing fine.

```

[Return to the Top](#)

Raw Strings

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```

1 >>> print(r'That is Carol\'s cat.')
2 That is Carol\''s cat.

```

Note: mostly used for regular expression definition (see `re` package)

[Return to the Top](#)

Multiline Strings with Triple Quotes

```

1 >>> print('''Dear Alice,
2 >>>
3 >>> Eve's cat has been arrested for catnapping, cat burglary, and extortion.
4 >>>
5 >>> Sincerely,
6 >>> Bob'''')
7 Dear Alice,
8
9 Eve's cat has been arrested for catnapping, cat burglary, and extortion.
10
11 Sincerely,
12 Bob

```

To keep a nicer flow in your code, you can use the `dedent` function from the `textwrap` standard package.

```

1 >>> from textwrap import dedent
2 >>>
3 >>> def my_function():
4 >>>     print('''
5 >>>         Dear Alice,
6 >>>
7 >>>         Eve's cat has been arrested for catnapping, cat burglary, and extortion.
8 >>>
9 >>>         Sincerely,
10 >>>         Bob
11 >>>     ''').strip()

```

This generates the same string than before.

[Return to the Top](#)

Indexing and Slicing Strings

```
1 H   e   l   l   o       w   o   r   l   d   !
2 0   1   2   3   4       5   6   7   8   9   10  11
```

```
1 >>> spam = 'Hello world!'
2
3 >>> spam[0]
4 'H'
```

```
1 >>> spam[4]
2 'o'
```

```
1 >>> spam[-1]
2 '!'
```

Slicing:

```
1 >>> spam[0:5]
2 'Hello'
```

```
1 >>> spam[:5]
2 'Hello'
```

```
1 >>> spam[6:]
2 'world!'
```

```
1 >>> spam[6:-1]
2 'world'
```

```
1 >>> spam[:-1]
2 'Hello world'
```

```
1 >>> spam[::-1]
2 '!dlrow olleH'
```

```
1 >>> spam = 'Hello world!'
2 >>> fizz = spam[0:5]
3 >>> fizz
4 'Hello'
```

[Return to the Top](#)

The in and not in Operators with Strings

```
1 >>> 'Hello' in 'Hello World'
2 True
```

```
1 >>> 'Hello' in 'Hello'
2 True
```

```
1 >>> 'HELLO' in 'Hello World'
2 False
```

```
1 >>> '' in 'spam'
2 True
```

```
1 >>> 'cats' not in 'cats and dogs'
2 False
```

The in and not in Operators with list

```
1 >>> a = [1, 2, 3, 4]
2 >>> 5 in a
3 False
```

```
1 >>> 2 in a
2 True
```

[Return to the Top](#)

The upper(), lower(), isupper(), and islower() String Methods

`upper()` and `lower()` :

```
1 >>> spam = 'Hello world!'
2 >>> spam = spam.upper()
3 >>> spam
4 'HELLO WORLD!'
```

```
1 >>> spam = spam.lower()
2 >>> spam
3 'hello world!'
```

isupper() and islower():

```
1 >>> spam = 'Hello world!'
2 >>> spam.islower()
3 False
```

```
1 >>> spam.isupper()
2 False
```

```
1 >>> 'HELLO'.isupper()
2 True
```

```
1 >>> 'abc12345'.islower()
2 True
```

```
1 >>> '12345'.islower()
2 False
```

```
1 >>> '12345'.isupper()
2 False
```

[Return to the Top](#)

The isX String Methods

- **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 new-lines 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.

[Return to the Top](#)

The startswith() and endswith() String Methods

```
1 >>> 'Hello world!'.startswith('Hello')
2 True
```

```
1 >>> 'Hello world!'.endswith('world!')
2 True
```

```
1 >>> 'abc123'.startswith('abcdef')
2 False
```

```
1 >>> 'abc123'.endswith('12')
2 False
```

```
1 >>> 'Hello world!'.startswith('Hello world!')
2 True
```

```
1 >>> 'Hello world!'.endswith('Hello world!')
2 True
```

[Return to the Top](#)

The join() and split() String Methods

join():

```
1 >>> ', '.join(['cats', 'rats', 'bats'])
2 'cats, rats, bats'
```

```
1 >>> ' '.join(['My', 'name', 'is', 'Simon'])
2 'My name is Simon'
```

```
1 >>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
2 'MyABCnameABCisABCSimon'
```

split():

```
1 >>> 'My name is Simon'.split()
2 ['My', 'name', 'is', 'Simon']
```

```
1 >>> 'MyABCnameABCisABCSimon'.split('ABC')
2 ['My', 'name', 'is', 'Simon']
```

```
1 >>> 'My name is Simon'.split('m')
2 ['My na', 'e is Si', 'on']
```

[Return to the Top](#)

Justifying Text with `rjust()`, `ljust()`, and `center()`

`rjust()` and `ljust()`:

```
1 >>> 'Hello'.rjust(10)
2      Hello'
```

```
1 >>> 'Hello'.rjust(20)
2      Hello'
```

```
1 >>> 'Hello World'.rjust(20)
2      Hello World'
```

```
1 >>> 'Hello'.ljust(10)
2 Hello      '
```

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
1 >>> 'Hello'.rjust(20, '*')
2 *****Hello'
```

```
1 >>> 'Hello'.ljust(20, '-')
2 Hello-----'
```

`center()`:

```
1 >>> 'Hello'.center(20)
2      Hello      '
```

```
1 >>> 'Hello'.center(20, '=')
2 =====Hello====='
```

[Return to the Top](#)

Removing Whitespace with `strip()`, `rstrip()`, and `lstrip()`

```
1 >>> spam = '    Hello World      '
2 >>> spam.strip()
3 'Hello World'
```

```
1 >>> spam.lstrip()
2 'Hello World '
```

```
1 >>> spam.rstrip()
2 '    Hello World'
```

```
1 >>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
2 >>> spam.strip('ampS')
3 'BaconSpamEggs'
```

[Return to the Top](#)

Copying and Pasting Strings with the pyperclip Module (need pip install)

```
1 >>> import pyperclip
2
3 >>> pyperclip.copy('Hello world!')
4
5 >>> pyperclip.paste()
6 'Hello world!'
```

[Return to the Top](#)

String Formatting

% operator

```
1 >>> name = 'Pete'
2 >>> 'Hello %s' % name
3 "Hello Pete"
```

We can use the `%x` format specifier to convert an int value to a string:

```
1 >>> num = 5
2 >>> 'I have %x apples' % num
3 "I have 5 apples"
```

Note: For new code, using `str.format` or `f-strings` (Python 3.6+) is strongly recommended over the `%` operator.

[Return to the Top](#)

String Formatting (`str.format`)

Python 3 introduced a new way to do string formatting that was later back-ported to Python 2.7. This makes the syntax for string formatting more regular.

```
1 >>> name = 'John'
```

```
2 >>> age = 20'
3
4 >>> "Hello I'm {}, my age is {}".format(name, age)
5 "Hello I'm John, my age is 20"
```

```
1 >>> "Hello I'm {0}, my age is {1}".format(name, age)
2 "Hello I'm John, my age is 20"
```

The official [Python 3.x documentation](#) recommend `str.format` over the `%` operator:

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the `str.format()` interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.

[Return to the Top](#)

Lazy string formatting

You would only use `%s` string formatting on functions that can do lazy parameters evaluation, the most common being logging:

Prefer:

```
1 >>> name = "alice"
2 >>> logging.debug("User name: %s", name)
```

Over:

```
>>> logging.debug("User name: {}".format(name))
```

Or:

```
>>> logging.debug("User name: " + name)
```

[Return to the Top](#)

Formatted String Literals or f-strings (Python 3.6+)

```
1 >>> name = 'Elizabeth'
2 >>> f'Hello {name}!'
3 'Hello Elizabeth!'
```

It is even possible to do inline arithmetic with it:

```
1 >>> a = 5
2 >>> b = 10
3 >>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
4 'Five plus ten is 15 and not 30.'
```

[Return to the Top](#)

Template Strings

A simpler and less powerful mechanism, but it is recommended when handling format strings generated by users. Due to their reduced complexity template strings are a safer choice.

```
1 >>> from string import Template
2 >>> name = 'Elizabeth'
3 >>> t = Template('Hey $name!')
4 >>> t.substitute(name=name)
5 'Hey Elizabeth!'
```

[Return to the Top](#)

Regular Expressions

1. Import the regex module with `import re`.
2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a `Match` object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

All the regex functions in Python are in the `re` module:

```
>>> import re
```

[Return to the Top](#)

Matching Regex Objects

```
1 >>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d\d\d\d\d')
2
3 >>> mo = phone_num_regex.search('My number is 415-555-4242.')
4
5 >>> print('Phone number found: {}'.format(mo.group()))
6 Phone number found: 415-555-4242
```

[Return to the Top](#)

Grouping with Parentheses

```
1 >>> phone_num_regex = re.compile(r'(\d\d\d)-(\d\d\d\d\d\d\d)')
2
3 >>> mo = phone_num_regex.search('My number is 415-555-4242.')
4
5 >>> mo.group(1)
6 '415'
7
8 >>> mo.group(2)
9 '555-4242'
10
11 >>> mo.group(0)
```

```
12 '415-555-4242'
13
14 >>> mo.group()
15 '415-555-4242'
```

To retrieve all the groups at once: use the `groups()` method—note the plural form for the name.

```
1 >>> mo.groups()
2 ('415', '555-4242')
3
4 >>> area_code, main_number = mo.groups()
5
6 >>> print(area_code)
7 415
8
9 >>> print(main_number)
10 555-4242
```

[Return to the Top](#)

Matching Multiple Groups with the Pipe

The `|` character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

```
1 >>> hero_regex = re.compile(r'Batman|Tina Fey')
2
3 >>> mo1 = hero_regex.search('Batman and Tina Fey.')
4
5 >>> mo1.group()
6 'Batman'
7
8 >>> mo2 = hero_regex.search('Tina Fey and Batman.')
9
10 >>> mo2.group()
11 'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex:

```
1 >>> bat_regex = re.compile(r'Bat(man|mobile|copter|bat)')
2
3 >>> mo = bat_regex.search('Batmobile lost a wheel')
4
5 >>> mo.group()
6 'Batmobile'
7
8 >>> mo.group(1)
9 'mobile'
```

[Return to the Top](#)

Optional Matching with the Question Mark

The `?` character flags the group that precedes it as an optional part of the pattern.

```
1 >>> bat_regex = re.compile(r'Bat(wo)?man')
2 >>> mo1 = bat_regex.search('The Adventures of Batman')
```

```
3 >>> mo1.group()
4 'Batman'
5
6 >>> mo2 = bat_regex.search('The Adventures of Batwoman')
7 >>> mo2.group()
8 'Batwoman'
```

[Return to the Top](#)

Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text.

```
1 >>> bat_regex = re.compile(r'Bat(wo)*man')
2 >>> mo1 = bat_regex.search('The Adventures of Batman')
3 >>> mo1.group()
4 'Batman'
5
6 >>> mo2 = bat_regex.search('The Adventures of Batwoman')
7 >>> mo2.group()
8 'Batwoman'
9
10 >>> mo3 = bat_regex.search('The Adventures of Batwowowowoman')
11 >>> mo3.group()
12 'Batwowowowoman'
```

[Return to the Top](#)

Matching One or More with the Plus

While * means “match zero or more,” the + (or plus) means “match one or more”. The group preceding a plus must appear at least once. It is not optional:

```
1 >>> bat_regex = re.compile(r'Bat(wo)+man')
2 >>> mo1 = bat_regex.search('The Adventures of Batwoman')
3 >>> mo1.group()
4 'Batwoman'
```

```
1 >>> mo2 = bat_regex.search('The Adventures of Batwowowowoman')
2 >>> mo2.group()
3 'Batwowowowoman'
```

```
1 >>> mo3 = bat_regex.search('The Adventures of Batman')
2 >>> mo3 is None
3 True
```

[Return to the Top](#)

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

```
1 >>> ha_regex = re.compile(r'(Ha){3}')
2 >>> mo1 = ha_regex.search('HaHaHa')
3 >>> mo1.group()
4 'HaHaHa'
```

```
1 >>> mo2 = ha_regex.search('Ha')
2 >>> mo2 is None
3 True
```

[Return to the Top](#)

Greedy and Nongreedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
1 >>> greedy_ha_regex = re.compile(r'(Ha){3,5}')
2 >>> mo1 = greedy_ha_regex.search('HaHaHaHaHa')
3 >>> mo1.group()
4 'HaHaHaHaHa'
```

```
1 >>> nongreedy_ha_regex = re.compile(r'(Ha){3,5}?)'
2 >>> mo2 = nongreedy_ha_regex.search('HaHaHaHaHa')
3 >>> mo2.group()
4 'HaHaHa'
```

[Return to the Top](#)

The `findall()` Method

In addition to the `search()` method, Regex objects also have a `findall()` method. While `search()` will return a `Match` object of the first matched text in the searched string, the `findall()` method will return the strings of every match in the searched string.

```
1 >>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d') # has no groups
2
3 >>> phone_num_regex.findall('Cell: 415-555-9999 Work: 212-555-0000')
4 ['415-555-9999', '212-555-0000']
```

To summarize what the `findall()` method returns, remember the following:

- When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d`, the method `findall()` returns a list of strings, such as `['415-555-9999', '212-555-0000']`.
- When called on a regex that has groups, such as `(\d\d\d)-(\d\d)-(\d\d\d)`, the method `findall()` returns a list of lists of strings (one string for each group), such as `[('415', '555', '9999'), ('212', '555', '0000')]`.

[Return to the Top](#)

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
1 >>> vowel_regex = re.compile(r'[^aeiouAEIOU]')
2
3 >>> vowel_regex.findall('Robocop eats baby food. BABY FOOD.')
4 ['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class. For example, enter the following into the interactive shell:

```
1 >>> consonant_regex = re.compile(r'[^aeiouAEIOU]')
2
3 >>> consonant_regex.findall('Robocop eats baby food. BABY FOOD.')
4 ['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

[Return to the Top](#)

The Caret and Dollar Sign Characters

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign (\$) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and \$ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

The r'^Hello' regular expression string matches strings that begin with 'Hello':

```
1 >>> begins_with_hello = re.compile(r'^Hello')
2
3 >>> begins_with_hello.search('Hello world!')
4 <_sre.SRE_Match object; span=(0, 5), match='Hello'>
5
6 >>> begins_with_hello.search('He said hello.') is None
7 True
```

The r'\d\$' regular expression string matches strings that end with a numeric character from 0 to 9:

```
1 >>> whole_string_is_num = re.compile(r'^\d+$')
2
3 >>> whole_string_is_num.search('1234567890')
4 <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
5
6 >>> whole_string_is_num.search('12345xyz67890') is None
7 True
8
9 >>> whole_string_is_num.search('12 34567890') is None
10 True
```

[Return to the Top](#)

The Wildcard Character

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline:

```
1 >>> at_regex = re.compile(r'.at')
2
3 >>> at_regex.findall('The cat in the hat sat on the flat mat.')
4 ['cat', 'hat', 'sat', 'lat', 'mat']
```

[Return to the Top](#)

Matching Everything with Dot-Star

```
1 >>> name_regex = re.compile(r'First Name: (.*) Last Name: (.*)')
2
3 >>> mo = name_regex.search('First Name: Al Last Name: Sweigart')
4
5 >>> mo.group(1)
6 'Al'
```

```
1 >>> mo.group(2)
2 'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?). The question mark tells Python to match in a nongreedy way:

```
1 >>> nongreedy_regex = re.compile(r'<.*?>')
2 >>> mo = nongreedy_regex.search('<To serve man> for dinner.>')
3 >>> mo.group()
4 '<To serve man>'
```

```
1 >>> greedy_regex = re.compile(r'<.*>')
2 >>> mo = greedy_regex.search('<To serve man> for dinner.>')
3 >>> mo.group()
4 '<To serve man> for dinner.>'
```

[Return to the Top](#)

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character:

```
1 >>> no_newline_regex = re.compile('.*')
2 >>> no_newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
3 'Serve the public trust.'
```

```
1 >>> newline_regex = re.compile('.*', re.DOTALL)
2 >>> newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
3 'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

[Return to the Top](#)

Review of Regex Symbols

Symbol	Matches
?	zero or one of the preceding group.
*	zero or more of the preceding group.
+	one or more of the preceding group.
{n}	exactly n of the preceding group.
{n,}	n or more of the preceding group.
{,m}	0 to m of the preceding group.
{n,m}	at least n and at most m of the preceding p.
{n,m}?	performs a nongreedy match of the preceding p.
^spam	means the string must begin with spam.
spam\$	means the string must end with spam.
.	any character, except newline characters.
\d , \w , and \s	a digit, word, or space character, respectively.
\D , \W , and \S	anything except a digit, word, or space, respectively.
[abc]	any character between the brackets (such as a, b,).
[^abc]	any character that isn't between the brackets.

[Return to the Top](#)

Case-Insensitive Matching

To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`:

```
1 >>> robocop = re.compile(r'robocop', re.I)
2
3 >>> robocop.search('Robocop is part man, part machine, all cop.').group()
4 'Robocop'
```

```
1 >>> robocop.search('ROBOCOP protects the innocent.').group()
2 'ROBOCOP'
```

```
1 >>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
```

```
2 'robocop'
```

[Return to the Top](#)

Substituting Strings with the sub() Method

The sub() method for Regex objects is passed two arguments:

1. The first argument is a string to replace any matches.
2. The second is the string for the regular expression.

The sub() method returns a string with the substitutions applied:

```
1 >>> names_regex = re.compile(r'Agent \w+')
2
3 >>> names_regex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
4 'CENSORED gave the secret documents to CENSORED.'
```

Another example:

```
1 >>> agent_names_regex = re.compile(r'Agent (\w)\w*')
2
3 >>> agent_names_regex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent')
4 ***** told C**** that E**** knew B**** was a double agent.'
```

[Return to the Top](#)

Managing Complex Regexes

To tell the re.compile() function to ignore whitespace and comments inside the regular expression string, “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

Now instead of a hard-to-read regular expression like this:

```
phone_regex = re.compile(r'((\d{3}|\(\d{3}\))?)?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s*(ext|x|ext.)\s*\d{2,5})?')'
```

you can spread the regular expression over multiple lines with comments like this:

```
1 phone_regex = re.compile(r'''(
2     (\d{3}|\(\d{3}\))?          # area code
3     (\s|-|\.)?                # separator
4     \d{3}                      # first 3 digits
5     (\s|-|\.)
6     \d{4}                      # last 4 digits
7     (\s*(ext|x|ext.)\s*\d{2,5})? # extension
8 )''', re.VERBOSE)
```

[Return to the Top](#)

Handling File and Directory Paths

There are two main modules in Python that deals with path manipulation. One is the `os.path` module and the other is the `pathlib` module. The `pathlib` module was added in Python 3.4, offering an object-oriented way to handle file system paths.

[Return to the Top](#)

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (`\`) as the separator between folder names. On Unix based operating system such as macOS, Linux, and BSDs, the forward slash (`/`) is used as the path separator. Joining paths can be a headache if your code needs to work on different platforms.

Fortunately, Python provides easy ways to handle this. We will showcase how to deal with this with both `os.path.join` and `pathlib.Path.joinpath`

Using `os.path.join` on Windows:

```
1 >>> import os
2
3 >>> os.path.join('usr', 'bin', 'spam')
4 'usr\\bin\\spam'
```

And using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2
3 >>> print(Path('usr').joinpath('bin').joinpath('spam'))
4 usr/bin/spam
```

`pathlib` also provides a shortcut to joinpath using the `/` operator:

```
1 >>> from pathlib import Path
2
3 >>> print(Path('usr') / 'bin' / 'spam')
4 usr/bin/spam
```

Notice the path separator is different between Windows and Unix based operating system, that's why you want to use one of the above methods instead of adding strings together to join paths together.

Joining paths is helpful if you need to create different file paths under the same directory.

Using `os.path.join` on Windows:

```
1 >>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
2
3 >>> for filename in my_files:
4     print(os.path.join('C:\\\\Users\\\\asweigart', filename))
5 C:\\Users\\asweigart\\accounts.txt
6 C:\\Users\\asweigart\\details.csv
7 C:\\Users\\asweigart\\invite.docx
```

Using `pathlib` on *nix:

```
1 >>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
2 >>> home = Path.home()
3 >>> for filename in my_files:
4 >>>     print(home / filename)
5 /home/asweigart/accounts.txt
6 /home/asweigart/details.csv
7 /home/asweigart/invite.docx
```

[Return to the Top](#)

The Current Working Directory

Using `os` on Windows:

```
1 >>> import os
2
3 >>> os.getcwd()
4 'C:\\Python34'
5 >>> os.chdir('C:\\Windows\\System32')
6
7 >>> os.getcwd()
8 'C:\\Windows\\System32'
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> from os import chdir
3
4 >>> print(Path.cwd())
5 /home/asweigart
6
7 >>> chdir('/usr/lib/python3.6')
8 >>> print(Path.cwd())
9 /usr/lib/python3.6
```

[Return to the Top](#)

Creating New Folders

Using `os` on Windows:

```
1 >>> import os
2 >>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> cwd = Path.cwd()
3 >>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir()
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "/usr/lib/python3.6/pathlib.py", line 1226, in mkdir
7     self._accessor.mkdir(self, mode)
8   File "/usr/lib/python3.6/pathlib.py", line 387, in wrapped
9     return strfunc(str(pathobj), *args)
```

```
10 FileNotFoundError: [Errno 2] No such file or directory: '/home/asweigart/delicious/walnut/waffles'
```

Oh no, we got a nasty error! The reason is that the 'delicious' directory does not exist, so we cannot make the 'walnut' and the 'waffles' directories under it. To fix this, do:

```
1 >>> from pathlib import Path
2 >>> cwd = Path.cwd()
3 >>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir(parents=True)
```

And all is good :)

[Return to the Top](#)

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's current working directory

There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

[Return to the Top](#)

Handling Absolute and Relative Paths

To see if a path is an absolute path:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isabs('/')
3 True
4 >>> os.path.isabs('..')
5 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('/').is_absolute()
3 True
4 >>> Path('..').is_absolute()
5 False
```

You can extract an absolute path with both `os.path` and `pathlib`

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.getcwd()
3 '/home/asweigart'
```

```
4 >>> os.path.abspath('..')
5 '/home'
```

Using `pathlib` on *nix:

```
1 from pathlib import Path
2 print(Path.cwd())
3 '/home/asweigart'
4 print(Path('..').resolve())
5 '/home'
```

You can get a relative path from a starting path to another path.

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.relpath('/etc/passwd', '/')
3 'etc/passwd'
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> print(Path('/etc/passwd').relative_to('/'))
3 etc/passwd
```

[Return to the Top](#)

Checking Path Validity

Checking if a file/directory exists:

Using `os.path` on *nix:

```
1 import os
2 >>> os.path.exists('.')
3 True
4 >>> os.path.exists('setup.py')
5 True
6 >>> os.path.exists('/etc')
7 True
8 >>> os.path.exists('nonexistentfile')
9 False
```

Using `pathlib` on *nix:

```
1 from pathlib import Path
2 >>> Path('.').exists()
3 True
4 >>> Path('setup.py').exists()
5 True
6 >>> Path('/etc').exists()
7 True
```

```
8 >>> Path('nonexistentfile').exists()
9 False
```

Checking if a path is a file:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isfile('setup.py')
3 True
4 >>> os.path.isfile('/home')
5 False
6 >>> os.path.isfile('nonexistentfile')
7 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('setup.py').is_file()
3 True
4 >>> Path('/home').is_file()
5 False
6 >>> Path('nonexistentfile').is_file()
7 False
```

Checking if a path is a directory:

Using `os.path` on *nix:

```
1 >>> import os
2 >>> os.path.isdir('/')
3 True
4 >>> os.path.isdir('setup.py')
5 False
6 >>> os.path.isdir('/spam')
7 False
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> Path('/').is_dir()
3 True
4 >>> Path('setup.py').is_dir()
5 False
6 >>> Path('/spam').is_dir()
7 False
```

[Return to the Top](#)

Finding File Sizes and Folder Contents

Getting a file's size in bytes:

Using `os.path` on Windows:

```
1 >>> import os
2 >>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
3 776192
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> stat = Path('/bin/python3.6').stat()
3 >>> print(stat) # stat contains some other information about the file as well
4 os.stat_result(st_mode=33261, st_ino=141087, st_dev=2051, st_nlink=2, st_uid=0,
5 --snip--
6 st_gid=0, st_size=10024, st_atime=1517725562, st_mtime=1515119809, st_ctime=1517261276)
7 >>> print(stat.st_size) # size in bytes
8 10024
```

Listing directory contents using `os.listdir` on Windows:

```
1 >>> import os
2 >>> os.listdir('C:\\Windows\\System32')
3 ['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
4 --snip--
5 'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Listing directory contents using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> for f in Path('/usr/bin').iterdir():
3 >>>     print(f)
4 ...
5 /usr/bin/tiff2rgba
6 /usr/bin/iconv
7 /usr/bin/ldd
8 /usr/bin/cache_restore
9 /usr/bin/udiskie
10 /usr/bin/unix2dos
11 /usr/bin/t1reencode
12 /usr/bin/epstopdf
13 /usr/bin/idle3
14 ...
```

To find the total size of all the files in this directory:

WARNING: Directories themselves also have a size! So you might want to check for whether a path is a file or directory using the methods in the methods discussed in the above section!

Using `os.path.getsize()` and `os.listdir()` together on Windows:

```
1 >>> import os
2 >>> total_size = 0
3
4 >>> for filename in os.listdir('C:\\Windows\\System32'):
5         total_size = total_size + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
6
```

```
7 >>> print(total_size)
8 1117846456
```

Using `pathlib` on *nix:

```
1 >>> from pathlib import Path
2 >>> total_size = 0
3
4 >>> for sub_path in Path('/usr/bin').iterdir():
5 ...     total_size += sub_path.stat().st_size
6 >>>
7 >>> print(total_size)
8 1903178911
```

Return to the Top

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders.

```
1 >>> import shutil, os
2
3 >>> os.chdir('C:\\\\')
4
5 >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
6     'C:\\delicious\\spam.txt'
7
8 >>> shutil.copy('eggs.txt', 'C:\\delicious\\\\eggs2.txt')
9     'C:\\delicious\\eggs2.txt'
```

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it:

```
1 >>> import shutil, os
2
3 >>> os.chdir('C:\\\\')
4
5 >>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
6     'C:\\bacon_backup'
```

Return to the Top

Moving and Renaming Files and Folders

```
1 >>> import shutil
2 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
3     'C:\\eggs\\bacon.txt'
```

The destination path can also specify a filename. In the following example, the source file is moved and renamed:

```
1 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\\\new_bacon.txt')
2     'C:\\eggs\\new_bacon.txt'
```

If there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

```
1 >>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
2 'C:\\eggs'
```

[Return to the Top](#)

Permanently Deleting Files and Folders

- Calling os.unlink(path) or Path.unlink() will delete the file at path.
- Calling os.rmdir(path) or Path.rmdir() will delete the folder at path. This folder must be empty of any files or folders.
- Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

[Return to the Top](#)

Safe Deletes with the send2trash Module

You can install this module by running pip install send2trash from a Terminal window.

```
1 >>> import send2trash
2
3 >>> with open('bacon.txt', 'a') as bacon_file: # creates the file
4     ...     bacon_file.write('Bacon is not a vegetable.')
5 25
6
7 >>> send2trash.send2trash('bacon.txt')
```

[Return to the Top](#)

Walking a Directory Tree

```
1 >>> import os
2 >>>
3 >>> for folder_name, subfolders, filenames in os.walk('C:\\delicious'):
4     ...     print('The current folder is {}'.format(folder_name))
5 >>>
6     ...     for subfolder in subfolders:
7         ...         print('SUBFOLDER OF {}: {}'.format(folder_name, subfolder))
8     ...     for filename in filenames:
9         ...         print('FILE INSIDE {}: {}'.format(folder_name, filename))
10    ...
11    ...     print('')
12 The current folder is C:\\delicious
13 SUBFOLDER OF C:\\delicious: cats
14 SUBFOLDER OF C:\\delicious: walnut
15 FILE INSIDE C:\\delicious: spam.txt
16
17 The current folder is C:\\delicious\\cats
18 FILE INSIDE C:\\delicious\\cats: catnames.txt
19 FILE INSIDE C:\\delicious\\cats: zophie.jpg
20
21 The current folder is C:\\delicious\\walnut
22 SUBFOLDER OF C:\\delicious\\walnut: waffles
23
24 The current folder is C:\\delicious\\walnut\\waffles
25 FILE INSIDE C:\\delicious\\walnut\\waffles: butter.txt
```

[Return to the Top](#)

`pathlib` provides a lot more functionality than the ones listed above, like getting file name, getting file extension, reading/writing a file without manually opening it, etc. Check out the [official documentation](#) if you want to know more!

Reading and Writing Files

The File Reading/Writing Process

To read/write to a file in Python, you will want to use the `with` statement, which will close the file for you after you are done.

[Return to the Top](#)

Opening and reading files with the `open()` function

```
1  >>> with open('C:\\\\Users\\\\your_home_folder\\\\hello.txt') as hello_file:
2  ...     hello_content = hello_file.read()
3  >>> hello_content
4  'Hello World!'
5
6  >>> # Alternatively, you can use the *readlines()* method to get a list of string values from the file, one string
7
8  >>> with open('sonnet29.txt') as sonnet_file:
9  ...     sonnet_file.readlines()
10 [When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
11 outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
12 look upon myself and curse my fate,']
13
14 >>> # You can also iterate through the file line by line:
15 >>> with open('sonnet29.txt') as sonnet_file:
16 ...     for line in sonnet_file: # note the new line character will be included in the line
17 ...         print(line, end='')
18
19 When, in disgrace with fortune and men's eyes,
20 I all alone beweep my outcast state,
21 And trouble deaf heaven with my bootless cries,
22 And look upon myself and curse my fate,
```

[Return to the Top](#)

Writing to Files

```
1  >>> with open('bacon.txt', 'w') as bacon_file:
2  ...     bacon_file.write('Hello world!\n')
3  13
4
5  >>> with open('bacon.txt', 'a') as bacon_file:
6  ...     bacon_file.write('Bacon is not a vegetable.')
7  25
8
9  >>> with open('bacon.txt') as bacon_file:
10 ...     content = bacon_file.read()
11
12 >>> print(content)
13 Hello world!
14 Bacon is not a vegetable.
```

[Return to the Top](#)

Saving Variables with the `shelve` Module

To save variables:

```
1 >>> import shelve
2
3 >>> cats = ['Zophie', 'Pooka', 'Simon']
4 >>> with shelve.open('mydata') as shelf_file:
5 ...     shelf_file['cats'] = cats
```

To open and read variables:

```
1 >>> with shelve.open('mydata') as shelf_file:
2 ...     print(type(shelf_file))
3 ...     print(shelf_file['cats'])
4 <class 'shelve.DbfilenameShelf'>
5 ['Zophie', 'Pooka', 'Simon']
```

Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form.

```
1 >>> with shelve.open('mydata') as shelf_file:
2 ...     print(list(shelf_file.keys()))
3 ...     print(list(shelf_file.values()))
4 ['cats']
5 [['Zophie', 'Pooka', 'Simon']]
```

[Return to the Top](#)

Saving Variables with the pprint.pformat() Function

```
1 >>> import pprint
2
3 >>> cats = [{name: 'Zophie', desc: 'chubby'}, {name: 'Pooka', desc: 'fluffy'}]
4
5 >>> pprint.pformat(cats)
6 "[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
7
8 >>> with open('myCats.py', 'w') as file_obj:
9 ...     file_obj.write('cats = {} \n'.format(pprint.pformat(cats)))
10 83
```

[Return to the Top](#)

Reading ZIP Files

```
1 >>> import zipfile, os
2
3 >>> os.chdir('C:\\\\')    # move to the folder with example.zip
4 >>> with zipfile.ZipFile('example.zip') as example_zip:
5 ...     print(example_zip.namelist())
6 ...     spam_info = example_zip.getinfo('spam.txt')
7 ...     print(spam_info.file_size)
8 ...     print(spam_info.compress_size)
9 ...     print('Compressed file is %sx smaller!' % (round(spam_info.file_size / spam_info.compress_size, 2)))
10
11 ['spam.txt', 'cats//', 'cats/catnames.txt', 'cats/zophie.jpg']
12 13908
13 3828
14 'Compressed file is 3.63x smaller!'
```

[Return to the Top](#)

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
1 >>> import zipfile, os
2
3 >>> os.chdir('C:\\\\')      # move to the folder with example.zip
4
5 >>> with zipfile.ZipFile('example.zip') as example_zip:
6     ...     example_zip.extractall()
```

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
1 >>> with zipfile.ZipFile('example.zip') as example_zip:
2     ...     print(example_zip.extract('spam.txt'))
3     ...     print(example_zip.extract('spam.txt', 'C:\\\\some\\\\new\\\\folders'))
4 'C:\\\\spam.txt'
5 'C:\\\\some\\\\new\\\\folders\\\\spam.txt'
```

[Return to the Top](#)

Creating and Adding to ZIP Files

```
1 >>> import zipfile
2
3 >>> with zipfile.ZipFile('new.zip', 'w') as new_zip:
4     ...     new_zip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
```

This code will create a new ZIP file named `new.zip` that has the compressed contents of `spam.txt`.

[Return to the Top](#)

JSON, YAML and configuration files

JSON

Open a JSON file with:

```
1 import json
2 with open("filename.json", "r") as f:
3     content = json.loads(f.read())
```

Write a JSON file with:

```
1 import json
2
3 content = {"name": "Joe", "age": 20}
4 with open("filename.json", "w") as f:
5     f.write(json.dumps(content, indent=2))
```

[Return to the Top](#)

YAML

Compared to JSON, YAML allows for much better human maintainability and gives you the option to add comments. It is a convenient choice for configuration files where humans will have to edit it.

There are two main libraries allowing to access to YAML files:

- [PyYaml](#)
- [Ruamel.yaml](#)

Install them using `pip install` in your virtual environment.

The first one is easier to use but the second one, Ruamel, implements much better the YAML specification, and allows for example to modify a YAML content without altering comments.

Open a YAML file with:

```
1 from ruamel.yaml import YAML
2
3 with open("filename.yaml") as f:
4     yaml=YAML()
5     yaml.load(f)
```

[Return to the Top](#)

Anyconfig

[Anyconfig](#) is a very handy package allowing to abstract completely the underlying configuration file format. It allows to load a Python dictionary from JSON, YAML, TOML, and so on.

Install it with:

```
pip install anyconfig
```

Usage:

```
1 import anyconfig
2
3 conf1 = anyconfig.load("/path/to/foo/conf.d/a.yaml")
```

[Return to the Top](#)

Debugging

Raising Exceptions

Exceptions are raised with a `raise` statement. In code, a `raise` statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function

- A string with a helpful error message passed to the `Exception()` function

```

1 >>> raise Exception('This is the error message.')
2 Traceback (most recent call last):
3   File "<pyshell#191>", line 1, in <module>
4     raise Exception('This is the error message.')
5 Exception: This is the error message.

```

Often it's the code that calls the function, not the function itself, that knows how to handle an exception. So you will commonly see a `raise` statement inside a function and the `try` and `except` statements in the code calling the function.

```

1 def box_print(symbol, width, height):
2     if len(symbol) != 1:
3         raise Exception('Symbol must be a single character string.')
4     if width <= 2:
5         raise Exception('Width must be greater than 2.')
6     if height <= 2:
7         raise Exception('Height must be greater than 2.')
8     print(symbol * width)
9     for i in range(height - 2):
10        print(symbol + (' ' * (width - 2)) + symbol)
11    print(symbol * width)
12 for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
13     try:
14         box_print(sym, w, h)
15     except Exception as err:
16         print('An exception happened: ' + str(err))

```

[Return to the Top](#)

Getting the Traceback as a String

The traceback is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

```

1 >>> import traceback
2
3 >>> try:
4 >>>     raise Exception('This is the error message.')
5 >>> except:
6 >>>     with open('errorInfo.txt', 'w') as error_file:
7 >>>         error_file.write(traceback.format_exc())
8 >>>     print('The traceback info was written to errorInfo.txt.')
9 116
10 The traceback info was written to errorInfo.txt.

```

The 116 is the return value from the `write()` method, since 116 characters were written to the file. The traceback text was written to `errorInfo.txt`.

```

1 Traceback (most recent call last):
2   File "<pyshell#28>", line 2, in <module>
3     Exception: This is the error message.

```

[Return to the Top](#)

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an `AssertionError` exception is raised. In code, an assert statement consists of the following:

- The `assert` keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

```
1 >>> pod_bay_door_status = 'open'
2
3 >>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
4
5 >>> pod_bay_door_status = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'
6
7 >>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
8
9 Traceback (most recent call last):
10   File "<pyshell#10>", line 1, in <module>
11     assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
12 AssertionError: The pod bay doors need to be "open".
```

In plain English, an assert statement says, “I assert that this condition holds true, and if not, there is a bug somewhere in the program.” Unlike exceptions, your code should not handle assert statements with `try` and `except`; if an assert fails, your program should crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that’s causing the bug.

Disabling Assertions

Assertions can be disabled by passing the `-O` option when running Python.

[Return to the Top](#)

Logging

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the `#!/usr/bin/python` shebang line):

```
1 import logging
2
3 logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
```

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is $1 \times 2 \times 3 \times 4$, or 24. Factorial 7 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as `factorialLog.py`.

```
1 >>> import logging
2 >>>
3 >>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
4 >>>
5 >>> logging.debug('Start of program')
6 >>>
7 >>> def factorial(n):
8 >>>
9 >>>     logging.debug('Start of factorial(%s)' % (n))
```

```

10 >>>     total = 1
11 >>>
12 >>>     for i in range(1, n + 1):
13 >>>         total *= i
14 >>>         logging.debug('i is ' + str(i) + ', total is ' + str(total))
15 >>>
16 >>>     logging.debug('End of factorial(%s)' % (n))
17 >>>
18 >>>     return total
19 >>>
20 >>> print(factorial(5))
21 >>> logging.debug('End of program')
22 2015-05-23 16:20:12,664 - DEBUG - Start of program
23 2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
24 2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
25 2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
26 2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
27 2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
28 2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
29 2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
30 2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
31 0
32 2015-05-23 16:20:12,684 - DEBUG - End of program

```

[Return to the Top](#)

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
DEBUG	<code>logging.debug()</code>	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	<code>logging.info()</code>	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	<code>logging.warning()</code>	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	<code>logging.error()</code>	Used to record an error that caused the program to fail to do something.
CRITICAL	<code>logging.critical()</code>	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

[Return to the Top](#)

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand.

```

1 >>> import logging
2
3 >>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -%(levelname)s - %(message)s')
4
5 >>> logging.critical('Critical error! Critical error!')
6 2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
7
8 >>> logging.disable(logging.CRITICAL)
9
10 >>> logging.critical('Critical error! Critical error!')

```

```
11
12 >>> logging.error('Error! Error!')
```

[Return to the Top](#)

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
1 import logging
2
3 logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
```

[Return to the Top](#)

Lambda Functions

This function:

```
1 >>> def add(x, y):
2         return x + y
3
4 >>> add(5, 3)
5 8
```

Is equivalent to the *lambda* function:

```
1 >>> add = lambda x, y: x + y
2 >>> add(5, 3)
3 8
```

It's not even need to bind it to a name like `add` before:

```
1 >>> (lambda x, y: x + y)(5, 3)
2 8
```

Like regular nested functions, lambdas also work as lexical closures:

```
1 >>> def make_adder(n):
2         return lambda x: x + n
3
4 >>> plus_3 = make_adder(3)
5 >>> plus_5 = make_adder(5)
6
7 >>> plus_3(4)
8 7
9 >>> plus_5(4)
10 9
```

Note: lambda can only evaluate an expression, like a single line of code.

[Return to the Top](#)

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
1 >>> age = 15
2
3 >>> print('kid' if age < 18 else 'adult')
4 kid
```

Ternary operators can be chained:

```
1 >>> age = 15
2
3 >>> print('kid' if age < 13 else 'teenager' if age < 18 else 'adult')
4 teenager
```

The code above is equivalent to:

```
1 if age < 18:
2     if age < 13:
3         print('kid')
4     else:
5         print('teenager')
6 else:
7     print('adult')
```

[Return to the Top](#)

args and kwargs

The names `args` and `kwargs` are arbitrary - the important thing are the `*` and `**` operators. They can mean:

1. In a function declaration, `*` means “pack all remaining positional arguments into a tuple named `<name>`”, while `**` is the same for keyword arguments (except it uses a dictionary, not a tuple).
2. In a function call, `*` means “unpack tuple or list named `<name>` to positional arguments at this position”, while `**` is the same for keyword arguments.

For example you can make a function that you can use to call any other function, no matter what parameters it has:

```
1 def forward(f, *args, **kwargs):
2     return f(*args, **kwargs)
```

Inside forward, args is a tuple (of all positional arguments except the first one, because we specified it - the f), kwargs is a dict. Then we call f and unpack them so they become normal arguments to f.

You use `*args` when you have an indefinite amount of positional arguments.

```
1 >>> def fruits(*args):
2 >>>     for fruit in args:
3 >>>         print(fruit)
4
5 >>> fruits("apples", "bananas", "grapes")
6
7 "apples"
8 "bananas"
9 "grapes"
```

Similarly, you use `**kwargs` when you have an indefinite number of keyword arguments.

```
1 >>> def fruit(**kwargs):
2 >>>     for key, value in kwargs.items():
3 >>>         print("{0}: {1}".format(key, value))
4
5 >>> fruit(name = "apple", color = "red")
6
7 name: apple
8 color: red
```

```
1 >>> def show(arg1, arg2, *args, kwarg1=None, kwarg2=None, **kwargs):
2 >>>     print(arg1)
3 >>>     print(arg2)
4 >>>     print(args)
5 >>>     print(kwarg1)
6 >>>     print(kwarg2)
7 >>>     print(kwargs)
8
9 >>> data1 = [1,2,3]
10 >>> data2 = [4,5,6]
11 >>> data3 = {'a':7,'b':8,'c':9}
12
13 >>> show(*data1,*data2, kwarg1="python",kwarg2="cheatsheet",**data3)
14 1
15 2
16 (3, 4, 5, 6)
17 python
18 cheatsheet
19 {'a': 7, 'b': 8, 'c': 9}
20
21 >>> show(*data1, *data2, **data3)
22 1
23 2
24 (3, 4, 5, 6)
25 None
26 None
27 {'a': 7, 'b': 8, 'c': 9}
28
29 # If you do not specify ** for kwargs
30 >>> show(*data1, *data2, *data3)
31 1
32 2
33 (3, 4, 5, 6, "a", "b", "c")
```

```
34 None
35 None
36 {}
```

Things to Remember(args)

1. Functions can accept a variable number of positional arguments by using `*args` in the def statement.
2. You can use the items from a sequence as the positional arguments for a function with the `*` operator.
3. Using the `*` operator with a generator may cause your program to run out of memory and crash.
4. Adding new positional parameters to functions that accept `*args` can introduce hard-to-find bugs.

Things to Remember(kwargs)

1. Function arguments can be specified by position or by keyword.
2. Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
3. Keyword arguments with default values make it easy to add new behaviors to a function, especially when the function has existing callers.
4. Optional keyword arguments should always be passed by keyword instead of by position.

[Return to the Top](#)

Context Manager

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes.

with statement

A context manager is an object that is notified when a context (a block of code) starts and ends. You commonly use one with the `with` statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
1 >>> with open(filename) as f:
2   ...     file_contents = f.read()
3
4 # the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's `exit` method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way.

Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
1 >>> import contextlib
2 >>> @contextlib.contextmanager
3 ... def context_manager(num):
4 ...     print('Enter')
5 ...     yield num + 1
6 ...     print('Exit')
7 >>> with context_manager(2) as cm:
```

```
8 ...      # the following instructions are run when the 'yield' point of the context
9 ...      # manager is reached.
10 ...     # 'cm' will have the value that was yielded
11 ...     print('Right in the middle with cm = {}'.format(cm))
12 Enter
13 Right in the middle with cm = 3
14 Exit
15
16 >>>
```

[Return to the Top](#)

__main__ Top-level script environment

`__main__` is the name of the scope in which top-level code executes. A module's **name** is set equal to `__main__` when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run as a script or with `python -m` but not when it is imported:

```
1 >>> if __name__ == "__main__":
2 ...     # execute only if run as a script
3 ...     main()
```

For a package, the same effect can be achieved by including a `main.py` module, the contents of which will be executed when the module is run with `-m`

For example we are developing script which is designed to be used as module, we should do:

```
1 >>> # Python program to execute function directly
2 >>> def add(a, b):
3 ...     return a+b
4 ...
5 >>> add(10, 20) # we can test it by calling the function save it as calculate.py
6 30
7 >>> # Now if we want to use that module by importing we have to comment out our call,
8 >>> # Instead we can write like this in calculate.py
9 >>> if __name__ == "__main__":
10 ...     add(3, 5)
11 ...
12 >>> import calculate
13 >>> calculate.add(3, 5)
14 8
```

Advantages

1. Every Python module has its `__name__` defined and if this is `__main__`, it implies that the module is being run standalone by the user and we can do corresponding appropriate actions.
2. If you import this script as a module in another script, the **name** is set to the name of the script/module.
3. Python files can act as either reusable modules, or as standalone programs.
4. if `__name__ == "main":` is used to execute some code only if the file was run directly, and not imported.

[Return to the Top](#)

setup.py

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing.

The `setup.py` file is at the heart of a Python project. It describes all of the metadata about your project. There are quite a few fields you can add to a project to give it a rich set of metadata describing the project. However, there are only three required fields: name, version, and packages. The name field must be unique if you wish to publish your package on the Python Package Index (PyPI). The version field keeps track of different releases of the project. The packages field describes where you've put the Python source code within your project.

This allows you to easily install Python packages. Often it's enough to write:

```
python setup.py install
```

and module will install itself.

Our initial `setup.py` will also include information about the license and will re-use the `README.txt` file for the `long_description` field. This will look like:

```
1 >>> from distutils.core import setup
2 >>> setup(
3 ...     name='pythonCheatsheet',
4 ...     version='0.1',
5 ...     packages=['pipenv',],
6 ...     license='MIT',
7 ...     long_description=open('README.txt').read(),
8 ... )
```

Find more information visit <http://docs.python.org/install/index.html>.

[Return to the Top](#)

Dataclasses

`Dataclasses` are python classes but are suited for storing data objects. This module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes.

Features

1. They store data and represent a certain data type. Ex: A number. For people familiar with ORMs, a model instance is a data object. It represents a specific kind of entity. It holds attributes that define or represent the entity.
2. They can be compared to other objects of the same type. Ex: A number can be greater than, less than, or equal to another number.

Python 3.7 provides a decorator `dataclass` that is used to convert a class into a dataclass.

`python 2.7`

```
1 >>> class Number:
2 ...     def __init__(self, val):
3 ...         self.val = val
4 ...
5 >>> obj = Number(2)
6 >>> obj.val
7 2
```

with dataclass

```
1 >>> @dataclass
2 ... class Number:
3 ...     val: int
4 ...
5 >>> obj = Number(2)
6 >>> obj.val
7 2
```

[Return to the Top](#)

Default values

It is easy to add default values to the fields of your data class.

```
1 >>> @dataclass
2 ... class Product:
3 ...     name: str
4 ...     count: int = 0
5 ...     price: float = 0.0
6 ...
7 >>> obj = Product("Python")
8 >>> obj.name
9 Python
10 >>> obj.count
11 0
12 >>> obj.price
13 0.0
```

Type hints

It is mandatory to define the data type in dataclass. However, If you don't want specify the datatype then, use `typing.Any`.

```
1 >>> from dataclasses import dataclass
2 >>> from typing import Any
3
4 >>> @dataclass
5 ... class WithoutExplicitTypes:
6 ...     name: Any
7 ...     value: Any = 42
8 ...
```

[Return to the Top](#)

Virtual Environment

The use of a Virtual Environment is to test python code in encapsulated environments and to also avoid filling the base Python installation with libraries we might use for only one project.

[Return to the Top](#)

virtualenv

1. Install virtualenv

```
pip install virtualenv
```

2. Install virtualenvwrapper-win (Windows)

```
pip install virtualenvwrapper-win
```

Usage:

1. Make a Virtual Environment

```
mkvirtualenv HelloWold
```

Anything we install now will be specific to this project. And available to the projects we connect to this environment.

2. Set Project Directory

To bind our virtualenv with our current working directory we simply enter:

```
setprojectdir .
```

3. Deactivate

To move onto something else in the command line type 'deactivate' to deactivate your environment.

```
deactivate
```

Notice how the parenthesis disappear.

4. Workon

Open up the command prompt and type 'workon HelloWold' to activate the environment and move into your root project folder

```
workon HelloWold
```

[Return to the Top](#)

poetry

Poetry is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

1. Install Poetry

```
pip install --user poetry
```

2. Create a new project

```
poetry new my-project
```

This will create a my-project directory:

```
1 my-project
2 └── pyproject.toml
3 └── README.rst
4 └── poetry_demo
5   └── __init__.py
6 └── tests
7   └── __init__.py
8     └── test_poetry_demo.py
```

The pyproject.toml file will orchestrate your project and its dependencies:

```
1 [tool.poetry]
2 name = "my-project"
3 version = "0.1.0"
4 description = ""
5 authors = ["your name <your@mail.com>"]
6
7 [tool.poetry.dependencies]
8 python = "*"
9
10 [tool.poetry.dev-dependencies]
11 pytest = "^3.4"
```

3. Packages

To add dependencies to your project, you can specify them in the tool.poetry.dependencies section:

```
1 [tool.poetry.dependencies]
2 pendulum = "^1.4"
```

Also, instead of modifying the pyproject.toml file by hand, you can use the add command and it will automatically find a suitable version constraint.

```
$ poetry add pendulum
```

To install the dependencies listed in the pyproject.toml:

```
poetry install
```

To remove dependencies:

```
poetry remove pendulum
```

For more information, check the [documentation](#).

[Return to the Top](#)

pipenv

Pipenv is a tool that aims to bring the best of all packaging worlds (bundler, composer, npm, cargo, yarn, etc.) to the Python world. Windows is a first-class citizen, in our world.

1. Install pipenv

```
pip install pipenv
```

2. Enter your Project directory and install the Packages for your project

```
1 cd my_project  
2 pipenv install <package>
```

Pipenv will install your package and create a Pipfile for you in your project's directory. The Pipfile is used to track which dependencies your project needs in case you need to re-install them.

3. Uninstall Packages

```
pipenv uninstall <package>
```

4. Activate the Virtual Environment associated with your Python project

```
pipenv shell
```

5. Exit the Virtual Environment

```
exit
```

Find more information and a video in [docs.pipenv.org](#).

[Return to the Top](#)

anaconda

Anaconda is another popular tool to manage python packages.

Where packages, notebooks, projects and environments are shared. Your place for free public conda package hosting.

Usage:

1. Make a Virtual Environment

```
conda create -n HelloWorld
```

2. To use the Virtual Environment, activate it by:

```
conda activate HelloWorld
```

Anything installed now will be specific to the project HelloWorld

3. Exit the Virtual Environment

```
conda deactivate
```

Code Signal CGA Sprint Resources



DATA_STRUC_PYTHON_NOTES-3

<https://replit.com/@bgoonz/DATASTRUCPYTHONNOTES-3#.replit>

CodeSignal GCA Info

Brian "Beej Jorgensen" Hall edited this page on Jun 3 · 24 revisions

The test is four problems, which are CodeSignal/HackerRank/Leetcode-style programming challenges. Problem 1 is generally the easiest, progressing to the hardest with problem 4.

The result score is between 300 and 850 (best). 650 is about the low end for Junior Dev.

This test is about your problem-solving skill, not about your Googling skill. You are expected to ponder the problems and come up with your own solution. Pasting solutions in from elsewhere and trying to make them work is prohibited. Even googling for another solution is prohibited. (You can, however, search for syntax help.)

For this reason, you should practice this skill throughout CS. **CS is all about using UPER to solve problems you've never seen before without Googling, and the GCA measures this skill.**

Background

Since GCA is designed to measure skills that are important for almost all software developers, CodeSignal has aimed to find the common denominator between three different sources of data.

1. What are the most common topics taught in different CS programs at 4-year Universities in the US?
2. What are the most common topics covered during technical interviews at successful US-based companies?
3. What are the most common questions asked on StackOverflow that are about general programming and not specialized domain knowledge?

CodeSignal has used MIT OCW, EdX, Coursera, and Udacity course catalogs as a source for #1. They've used the book *Cracking the Coding Interview*, CodeSignal Interview Practice Mode, Leetcode, CareerCup, and Glassdoor Interview Questions sections to identify #2. And StackOverflow public API for #3.

How to Practice

Do the coursework in Lambda CS.

- Try to generally restrict your searching to syntax unless otherwise directed
- Don't search for problem solutions for the sprint challenges; syntax only
- Same for the GCA, proctor-enforced

Additional, optional resources:

- Go to the [CodeSignal Arcade](#) and solve questions in *The Core* without looking up the answer. If you look up the answer, it doesn't count.
 - Solve the first 50 problems
 - Except spend your earned coins to skip *Corner of 0s and 1s*
 - Some of these are challenging—feel free to buy your way ahead if you need to come back to a hard problem later

- Keep solving more for more practice.
- Once you get a few hundred coins, unlock the interview practice and other things in the main menu.
- Head over to Leetcode and tackle the easy and medium [leetcode algorithms](#) problems, or problems on your site of choice.
- Take the [GCA Practice Test](#) (24-hour cooldown).

Scoring

If you get 100% of the tests passing on a submission for a set of problems, you'll get a base score, listed in the table below. This base score is modified up or down based on a variety of factors.

Score Modifiers

The score is modified ± 12 points based on three factors:

- Number of attempts
 - Your score will be modded if you make more or fewer than the average number of submissions for a particular problem.
 - Running the tests doesn't count as a submission. You must click the `submit` button for it to count.
- Time taken
 - Your score will be modded if you take longer or shorter than the average amount of time to solve a problem.
- Code quality
 - Your code quality will be automatically judged based on a variety of factors, e.g.: consistent spacing and indentation.

Again, the most the base score will change as determined by these factors is ± 12 points.

Strategy recommendation: don't worry about the modifiers. Just concentrate on UPER and solving the problem.

Partial Credit

If you submit and pass fewer than 100% of the tests, you'll receive partial credit for the submission depending on how many tests you did pass. In other words, you won't achieve the base score for that problem, but will get part way toward the base score.

Scoring Table

If you get 100% of tests passing for the listed questions in the left column, your base score is in the center. On the right is the modifier range, ± 12 points from the base score.

Correct Answers	Base Score	Modifier Range
1 . . .	662	650-674
. 2 . .	700	688-712
1 2 . .	712	700-724
. . 3 .	731	719-743
1 . 3 .	743	731-755
. 2 3 .	750	738-762
1 2 3 .	763	751-775
. . . 4	780	768-792
1 . . 4	792	780-804
. 2 . 4	799	787-811
. . 3 4	805	793-817

1 2 . 4	812	800-824
1 . 3 4	818	806-830
. 2 3 4	825	813-837
1 2 3 4	837	825-849

For example, if you get questions 1, 2, and 3 100% correct, but you make a lot of incorrect submissions and take longer than average, you'll score closer to 751. If you get them correct in the first submission in record time, you'll score closer to 775.

Strategy recommendation: Choose the problem that looks the easiest to tackle first. This is likely question 1, but read them all to find out.

Implementation, Problem-Solving, and Speed Ratings

In addition to the numeric score, additional ratings are presented. Lambda does not use these ratings, but they are included for your information.

Speed is your rating compared to the average speed to solve the problems.

The other two ratings are determined by which problems are solved, as shown below:

Solved Tasks	Implementation	Problem-Solving
1 . . .	Low	Low
. 2 . .	Fair	Fair
1 2 . .	Fair	Fair
1 2 3 .	Good	Average
1 2 . 4	Good	Good
1 2 3 4	Excellent	Excellent

Question 1

NOTE: these aren't definitive or complete lists! They don't say exactly what will be on the test, and the test questions might require more or less knowledge than listed. The information below is included to give you an idea of the relative difficulty of each question.

Expected Knowledge

- Working with numbers.
 - Basic operations with numbers.
- Basic string manipulation.
 - Splitting a string into substrings.
 - Modifying the elements of a string.
- Basic array manipulation.
 - Iterating over an array.

Can Include

- Tasks that require a combination of 2 to 3 basic concepts. For example:
 - Iterating over an array and taking into account some condition.
 - Splitting a string by some condition.
- Should usually be solvable using one loop.

- The task description should clearly state the implementation steps.

Question 2

Expected Knowledge

- Working with numbers.
 - Basic operations with numbers.
 - Splitting numbers into digits.
- Basic string manipulation.
 - Splitting a string into substrings.
 - Comparing strings.
 - Modifying the elements of a string. – Concatenating strings.
 - Reversing a string.
- Basic array manipulation.
 - Iterating over an array.
 - Modifying the elements of an array.
 - Reversing an array.

Can Include

- Tasks that require a combination of 3 to 5 basic concepts. For example:
 - Splitting a string into substrings, modifying each substring and comparing with other strings.
 - Iterating over an array to produce two new arrays given some conditions, modifying the second array and appending it to the beginning of the first array.
- Should usually be solvable using one to two nested loops.
- The task description should clearly state the implementation steps.

Question 3

Expected Knowledge

- Includes everything from the previous task.
- Splitting a task into smaller subtasks/functions.
- Manipulating two-dimensional arrays.
 - Iterating over the elements in a particular order.
 - Modifying values.
 - Swapping rows/columns.
- Using hashmaps.
 - Using built in hashmaps to store strings or integers as keys.

Can Include

- Implementing a specific comparator for strings.
- Implementing a specific merge function for arrays.
- Other implementation challenges that clearly explain what needs to be done and require translating the instructions into code.

Question 4

Expected Knowledge

- Includes everything from previous tasks.
- Working with trees.
 - Storing and traversing trees.
 - Transforming trees.
- Understanding hashmaps.

- Solving tasks that require understanding the implementation of hashmaps.
- Fundamentals of discrete mathematics.
- Brute-force search.
 - Checking all possible solutions to find the optimal solution.

Can Include

- Tasks that require noticing an application of a certain algorithm, data-structure or technique.
- Optimizing some queries with the help of data structures like hashmaps or sets.
- Algorithms on trees like finding the longest path of a tree.

Rules

- [GCA Setup and Proctoring Rules](#)
- Additional clarifications to the rules:
 - Using scratch paper is allowed, but make it obvious (and maybe even say) that you're scribbling.
 - Using an off-screen whiteboard is allowed, but make it obvious that's what you're doing.
 - Recommend *against* referring to written notes since the proctor won't know what you're looking at.
 - Scan or take photos of your relevant notes so you can view them on-screen.
 - Prohibited: PythonTutor or any other external IDE, editor, debugger, or environment.
 - Exception: You **may** open a simple Python REPL in a terminal to quickly test commands or look up syntax.

Testing Link

- **SIGN IN WITH YOUR LAMBDASTUDENTS EMAIL!** If you're not sure, go to your CodeSignal profile and make sure it's set as your primary email.
 - **IF YOU DON'T, YOUR ATTEMPT WON'T COUNT!**
- There is a 2-week cooldown (measured down to the minute of your previous submission).
- [Take the GCA Now](#)
 - If this link fails, DM `@Beej` on Slack to get it updated.

See Also

- [What to Expect on the GCA](#) including a link to the practice test
- [Lambda's GCA HOWTO and FAQ](#)

Youtube

 CS47 Intro to Python I w/ Tom Tarpey

<https://youtu.be/ocxkkzjdFeY>

 CS47 Python II w/ Tom Tarpey

<https://youtu.be/j5Hu08FhAJQ>

 CS47 Python III w/ Tom Tarpey

<https://youtu.be/rxYrTtxefjE>

 CS47 Strings and Arrays Python IV w/ Tom Tarpey

<https://youtu.be/AKDIKZ6zwmw>

 CS46 Arrays and Strings Python IV w/ Tom Tarpey

<https://youtu.be/BJ8YtWWFUuw>

 CS46 Number Bases and Character Encoding w/ Tom Tarpey

<https://youtu.be/7bxLc0qwL2c>

 CS46 Hash Tables I w/ Tom Tarpey

<https://youtu.be/mYu3vNKp8SQ>



CS46 Hash Tables II w/ Tom Tarpey

<https://youtu.be/-S5CwtII718>



CS46 Searching & Recursion w/ Tom Tarpey

<https://youtu.be/MQmxFxERdCY>



CS46 Linked Lists I w/ Tom Tarpey

<https://youtu.be/PC0w44UH7Mo>



CS46 Stacks and Queues w/ Tom Tarpey

<https://youtu.be/qPgANRuDdhw>



CS46 Tree Traversals w/ Tom Tarpey

<https://youtu.be/BsWLku3l5ik>



CS46 Graphs I w/ Tom Tarpey

<https://youtu.be/KhR06pnMSCQ>



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******

Useful Links

- Getting started with python
- Installing Python3
- Running python programs
- Datatype & Variables
- Python numbers
- Python Strings
- Python Lists
- Python Dictionaries
- Python Tuples
- Datatype conversion
- Python Control Statements
- Python Functions
- Python Loops
- Python Mathematical Function
- Python Generating Random numbers
- Python File Handling
- Python Object and Classes
- Python Operator Overloading
- Python inheritance and polymorphism
- Python Exception Handling
- Python Modules

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-datasience](#)
- [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](#)

My-Links



GitHub: Web-Dev-Collaborative/jupyter-learn-py/aa9ad2b2ffa97ac278c13ebca6b89e4c1d1aad08

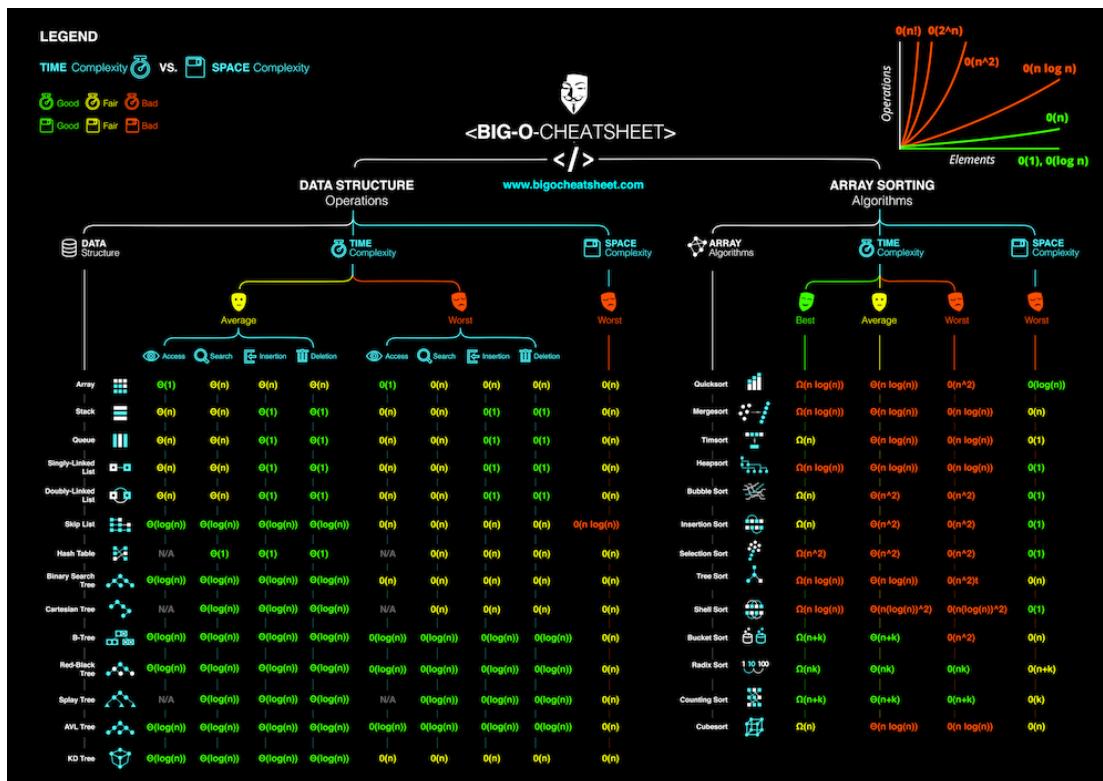
<https://mybinder.org/v2/gh/Web-Dev-Collaborative/jupyter-learn-py/aa9ad2b2ffa97ac278c13ebca6b89e4c1d1aad08>

Beginners Guide To Python

quick-reference

Useful Info

Big O Cheat Sheet



[bigO - CodeSandbox](https://codesandbox.io/s/3wqy4)

<https://codesandbox.io/s/3wqy4>

Built-in Functions:

[Built-in Functions – Python 3.9.7 documentation](https://docs.python.org/3/library/functions.html#dir)

<https://docs.python.org/3/library/functions.html#dir>

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

<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>

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()	

Python Glossary

This page is meant to be a quick reference guide to Python. If you see something that needs to be added, please let us know and we will add it to the list.

>>>

The default Python prompt of the interactive shell. Often seen for [code examples](#) which can be executed interactively in the interpreter.

abs

Return the absolute value of a number.

argparse

Argparse is a parser for command-line options, arguments and subcommands.

assert

Used during debugging to check for conditions that ought to apply

assignment

Giving a value to a variable.

block

Section of code which is grouped together

break Used to exit a for loop or a while loop.

class

A template for creating user-defined objects.

compiler

Translates a program written in a high-level language into a low-level language.

continue Used to skip the current block, and return to the “for” or “while” statement

conditional statement

Statement that contains an “if” or “if/else”.

debugging

The process of finding and removing programming errors.

def

Defines a function or method

dictionary

A mutable associative array (or dictionary) of key and value pairs. Can contain mixed types (keys and values). Keys must be a hashable type.

distutils

Package included in the Python Standard Library for installing, building and distributing Python code.

docstring

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.

future

A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

easy_install

Easy Install is a python module (`easy_install`) bundled with setuptools that lets you automatically download, build, install, and manage Python packages.

evaluation order

[Python evaluates expressions](#) from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

exceptions

Means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions

expression

Python code that produces a value.

filter

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true

float

An immutable floating point number.

for

Iterates over an iterable object, capturing each element to a local variable for use by the attached block

function

A parameterized sequence of statements.

function call

An invocation of the function with arguments.

garbage collection

The process of freeing memory when it is not used anymore.

generators

A function which returns an iterator.

high level language

Designed to be easy for humans to read and write.

IDLE

Integrated development environment

if statement

Conditionally executes a block of code, along with else and elif (a contraction of else-if).

immutable

Cannot be changed after its created.

import

Used to import modules whose functions or variables can be used in the current program.

indentation

Python uses white-space indentation, rather than curly braces or keywords, to delimit blocks.

int

An immutable integer of unlimited magnitude.

interactive mode

When commands are read from a tty, the interpreter is said to be in interactive mode.

interpret

Execute a program by translating it one line at a time.

IPython

Interactive shell for interactive computing.

Related: [iPython tutorial](#)

iterable

An object capable of returning its members one at a time.

lambda

They are a shorthand to create anonymous functions.

list

Mutable list, can contain mixed types.

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results.

literals

Literals are notations for constant values of some built-in types.

map

`map(function, iterable, ...)` Apply function to every item of iterable and return a list of the results.

methods

A method is like a function, but it runs “on” an object.

module

The basic unit of code reusability in Python. A block of code imported by some other code.

object

Any data with state (attributes or value) and defined behavior (methods).

object-oriented

allows users to manipulate data structures called objects in order to build and execute programs.

pass

Needed to create an empty code block

PEP 8

A set of recommendations how to write Python code.

Python Package Index

Official repository of third-party software for Python

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.

reduce

`reduce(function, sequence)` returns a single value constructed by calling the (binary) function on the first two items of the sequence, then on the result and the next item, and so on.

set

Unordered set, contains no duplicates

setuptools

Collection of enhancements to the Python distutils that allow you to more easily build and distribute Python packages

slice

Sub parts of sequences

str

A character string: an immutable sequence of Unicode codepoints.

strings

Can include numbers, letters, and various symbols and be enclosed by either double or single quotes, although single quotes are more commonly used.

statement

A statement is part of a suite (a “block” of code).

try

Allows exceptions raised in its attached code block to be caught and handled by except clauses.

tuple

Immutable, can contain mixed types.

variables

Placeholder for texts and numbers. The equal sign (=) is used to assign values to variables.

while

Executes a block of code as long as its condition is true.

with

Encloses a code block within a context manager.

yield

Returns a value from a generator function.

Zen of Python

When you type “import this”, Python’s philosophy is printed out.

index

Calculates the date of `n` days from the given date.

- Use `datetime.timedelta` and the `+` operator to calculate the new `datetime.datetime` value after adding `n` days to `d`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime, timedelta
2
3 def add_days(n, d = datetime.today()):
4     return d + timedelta(n)
```

```
1 from datetime import date
2
3 add_days(5, date(2020, 10, 25)) # date(2020, 10, 30)
4 add_days(-5, date(2020, 10, 25)) # date(2020, 10, 20)
```

Checks if all elements in a list are equal.

- Use `set()` to eliminate duplicate elements and then use `len()` to check if length is `1`.

```
1 def all_equal(lst):
2     return len(set(lst)) == 1
```

```
1 all_equal([1, 2, 3, 4, 5, 6]) # False
2 all_equal([1, 1, 1, 1]) # True
```

Checks if all the values in a list are unique.

- Use `set()` on the given list to keep only unique occurrences.
- Use `len()` to compare the length of the unique values to the original list.

```
1 def all_unique(lst):
2     return len(lst) == len(set(lst))
```

```
1 x = [1, 2, 3, 4, 5, 6]
2 y = [1, 2, 2, 3, 4, 5]
3 all_unique(x) # True
4 all_unique(y) # False
```

Generates a list of numbers in the arithmetic progression starting with the given positive integer and up to the specified limit.

- Use `range()` and `list()` with the appropriate start, step and end values.

```
1 def arithmetic_progression(n, lim):
2     return list(range(n, lim + 1, n))
```

```
arithmetic_progression(5, 25) # [5, 10, 15, 20, 25]
```

Calculates the average of two or more numbers.

- Use `sum()` to sum all of the `args` provided, divide by `len()`.

```
1 def average(*args):
2     return sum(args, 0.0) / len(args)
```

```
1 average(*[1, 2, 3]) # 2.0
2 average(1, 2, 3) # 2.0
```

Calculates the average of a list, after mapping each element to a value using the provided function.

- Use `map()` to map each element to the value returned by `fn`.
- Use `sum()` to sum all of the mapped values, divide by `len(lst)`.
- Omit the last argument, `fn`, to use the default identity function.

```
1 def average_by(lst, fn = lambda x: x):
2     return sum(map(fn, lst), 0.0) / len(lst)
```

```
1 average_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda x: x['n'])
2 # 5.0
```

Splits values into two groups, based on the result of the given `filter` list.

- Use a list comprehension and `zip()` to add elements to groups, based on `filter`.
- If `filter` has a truthy value for any element, add it to the first group, otherwise add it to the second group.

```
1 def bifurcate(lst, filter):
2     return [
3         [x for x, flag in zip(lst, filter) if flag],
4         [x for x, flag in zip(lst, filter) if not flag]
5     ]
```

```
1 bifurcate(['beep', 'boop', 'foo', 'bar'], [True, True, False, True])
2 # [['beep', 'boop', 'bar'], ['foo']]
```

Splits values into two groups, based on the result of the given filtering function.

- Use a list comprehension to add elements to groups, based on the value returned by `fn` for each element.
- If `fn` returns a truthy value for any element, add it to the first group, otherwise add it to the second group.

```
1 def bifurcate_by(lst, fn):
2     return [
3         [x for x in lst if fn(x)],
4         [x for x in lst if not fn(x)]
5     ]
```

```
1 bifurcate_by(['beep', 'boop', 'foo', 'bar'], lambda x: x[0] == 'b')
2 # [['beep', 'boop', 'bar'], ['foo']]
```

Calculates the number of ways to choose `k` items from `n` items without repetition and without order.

- Use `math.comb()` to calculate the binomial coefficient.

```
1 from math import comb
2
3 def binomial_coefficient(n, k):
4     return comb(n, k)
```

```
binomial_coefficient(8, 2) # 28
```

Returns the length of a string in bytes.

- Use `str.encode('utf-8')` to encode the given string and return its length.

```
1 def byte_size(s):
2     return len(s.encode('utf-8'))
```

```
1 byte_size(' ') # 4
2 byte_size('Hello World') # 11
```

Converts a string to camelcase.

- Use `re.sub()` to replace any `-` or `_` with a space, using the regexp `r"(_|-)+"`.
- Use `str.title()` to capitalize the first letter of each word and convert the rest to lowercase.
- Finally, use `str.replace()` to remove spaces between words.

```
1 from re import sub
2
3 def camel(s):
4     s = sub(r"(_|-)+", " ", s).title().replace(" ", "")
5     return ''.join([s[0].lower(), s[1:]])
```

```
1 camel('some_database_field_name') # 'someDatabaseFieldName'
2 camel('Some label that needs to be camelized')
3 # 'someLabelThatNeedsToBeCamelized'
4 camel('some-javascript-property') # 'someJavascriptProperty'
5 camel('some-mixed_string with spaces_underscores-and-hyphens')
6 # 'someMixedStringWithSpacesUnderscoresAndHyphens'
```

Capitalizes the first letter of a string.

- Use list slicing and `str.upper()` to capitalize the first letter of the string.
- Use `str.join()` to combine the capitalized first letter with the rest of the characters.
- Omit the `lower_rest` parameter to keep the rest of the string intact, or set it to `True` to convert to lowercase.

```
1 def capitalize(s, lower_rest = False):
2     return ''.join([s[:1].upper(), (s[1:]).lower() if lower_rest else s[1:]])
```

```
1 capitalize('fooBar') # 'FooBar'
2 capitalize('fooBar', True) # 'Foobar'
```

Capitalizes the first letter of every word in a string.

- Use `str.title()` to capitalize the first letter of every word in the string.

```
1 def capitalize_every_word(s):
2     return s.title()
```

```
capitalize_every_word('hello world!') # 'Hello World!'
```

Casts the provided value as a list if it's not one.

- Use `isinstance()` to check if the given value is enumerable.
- Return it by using `list()` or encapsulated in a list accordingly.

```
1 def cast_list(val):
2     return list(val) if isinstance(val, (tuple, list, set, dict)) else [val]
```

```
1 cast_list('foo') # ['foo']
2 cast_list([1]) # [1]
3 cast_list(('foo', 'bar')) # ['foo', 'bar']
```

unlisted: true

Converts Celsius to Fahrenheit.

- Follow the conversion formula $F = 1.8 * C + 32$.

```
1 def celsius_to_fahrenheit(degrees):
2     return ((degrees * 1.8) + 32)
```

```
celsius_to_fahrenheit(180) # 356.0
```

Creates a function that will invoke a predicate function for the specified property on a given object.

- Return a `lambda` function that takes an object and applies the predicate function, `fn` to the specified property.

```
1 def check_prop(fn, prop):
2     return lambda obj: fn(obj[prop])
```

```
1 check_age = check_prop(lambda x: x >= 18, 'age')
2 user = {'name': 'Mark', 'age': 18}
3 check_age(user) # True
```

Chunks a list into smaller lists of a specified size.

- Use `list()` and `range()` to create a list of the desired `size`.
- Use `map()` on the list and fill it with splices of the given list.
- Finally, return the created list.

```
1 from math import ceil
2
3 def chunk(lst, size):
4     return list(
5         map(lambda x: lst[x * size:x * size + size],
6             list(range(ceil(len(lst) / size)))))
```

```
chunk([1, 2, 3, 4, 5], 2) # [[1, 2], [3, 4], [5]]
```

Chunks a list into `n` smaller lists.

- Use `math.ceil()` and `len()` to get the size of each chunk.
- Use `list()` and `range()` to create a new list of size `n`.
- Use `map()` to map each element of the new list to a chunk the length of `size`.
- If the original list can't be split evenly, the final chunk will contain the remaining elements.

```
1 from math import ceil
2
3 def chunk_into_n(lst, n):
4     size = ceil(len(lst) / n)
5     return list(
6         map(lambda x: lst[x * size:x * size + size],
```

```
7     list(range(n)))
8 )
```

```
chunk_into_n([1, 2, 3, 4, 5, 6, 7], 4) # [[1, 2], [3, 4], [5, 6], [7]]
```

Clamps `num` within the inclusive range specified by the boundary values.

- If `num` falls within the range (`a`, `b`), return `num`.
- Otherwise, return the nearest number in the range.

```
1 def clamp_number(num, a, b):
2     return max(min(num, max(a, b)), min(a, b))
```

```
1 clamp_number(2, 3, 5) # 3
2 clamp_number(1, -1, -5) # -1
```

Inverts a dictionary with non-unique hashable values.

- Create a `collections.defaultdict` with `list` as the default value for each key.
- Use `dictionary.items()` in combination with a loop to map the values of the dictionary to keys using `dict.append()`.
- Use `dict()` to convert the `collections.defaultdict` to a regular dictionary.

```
1 from collections import defaultdict
2
3 def collect_dictionary(obj):
4     inv_obj = defaultdict(list)
5     for key, value in obj.items():
6         inv_obj[value].append(key)
7     return dict(inv_obj)
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 10,
4     'Anna': 9,
5 }
6 collect_dictionary(ages) # { 10: ['Peter', 'Isabel'], 9: ['Anna'] }
```

Combines two or more dictionaries, creating a list of values for each key.

- Create a new `collections.defaultdict` with `list` as the default value for each key and loop over `dicts`.
- Use `dict.append()` to map the values of the dictionary to keys.
- Use `dict()` to convert the `collections.defaultdict` to a regular dictionary.

```
1 from collections import defaultdict
2
3 def combine_values(*dicts):
4     res = defaultdict(list)
5     for d in dicts:
6         for key in d:
```

```
7     res[key].append(d[key])
8
9
10    return dict(res)
```

```
1 d1 = {'a': 1, 'b': 'foo', 'c': 400}
2 d2 = {'a': 3, 'b': 200, 'd': 400}
3
4 combine_values(d1, d2) # {'a': [1, 3], 'b': ['foo', 200], 'c': [400], 'd': [400]}
```

Removes falsy values from a list.

- Use `filter()` to filter out falsy values (`False`, `None`, `0`, and `""`).

```
1 def compact(lst):
2     return list(filter(None, lst))
```

```
compact([0, 1, False, 2, '', 3, 'a', 's', 34]) # [ 1, 2, 3, 'a', 's', 34 ]
```

Performs right-to-left function composition.

- Use `functools.reduce()` to perform right-to-left function composition.
- The last (rightmost) function can accept one or more arguments; the remaining functions must be unary.

```
1 from functools import reduce
2
3 def compose(*fns):
4     return reduce(lambda f, g: lambda *args: f(g(*args)), fns)
```

```
1 add5 = lambda x: x + 5
2 multiply = lambda x, y: x * y
3 multiply_and_add_5 = compose(add5, multiply)
4 multiply_and_add_5(5, 2) # 15
```

Performs left-to-right function composition.

- Use `functools.reduce()` to perform left-to-right function composition.
- The first (leftmost) function can accept one or more arguments; the remaining functions must be unary.

```
1 from functools import reduce
2
3 def compose_right(*fns):
4     return reduce(lambda f, g: lambda *args: g(f(*args)), fns)
```

```
1 add = lambda x, y: x + y
2 square = lambda x: x * x
3 add_and_square = compose_right(add, square)
4 add_and_square(1, 2) # 9
```

Groups the elements of a list based on the given function and returns the count of elements in each group.

- Use `collections.defaultdict` to initialize a dictionary.
- Use `map()` to map the values of the given list using the given function.
- Iterate over the map and increase the element count each time it occurs.

```
1 from collections import defaultdict
2
3 def count_by(lst, fn = lambda x: x):
4     count = defaultdict(int)
5     for val in map(fn, lst):
6         count[val] += 1
7     return dict(count)
```

```
1 from math import floor
2
3 count_by([6.1, 4.2, 6.3], floor) # {6: 2, 4: 1}
4 count_by(['one', 'two', 'three'], len) # {3: 2, 5: 1}
```

Counts the occurrences of a value in a list.

- Use `list.count()` to count the number of occurrences of `val` in `lst`.

```
1 def count_occurrences(lst, val):
2     return lst.count(val)
```

```
count_occurrences([1, 1, 2, 1, 2, 3], 1) # 3
```

Creates a list of partial sums.

- Use `itertools.accumulate()` to create the accumulated sum for each element.
- Use `list()` to convert the result into a list.

```
1 from itertools import accumulate
2
3 def cumsum(lst):
4     return list(accumulate(lst))
```

```
cumsum(range(0, 15, 3)) # [0, 3, 9, 18, 30]
```

Curries a function.

- Use `functools.partial()` to return a new partial object which behaves like `fn` with the given arguments, `args`, partially applied.

```
1 from functools import partial
2
3 def curry(fn, *args):
```

```
4     return partial(fn, *args)
```

```
1 add = lambda x, y: x + y
2 add10 = curry(add, 10)
3 add10(20) # 30
```

Creates a list of dates between `start` (inclusive) and `end` (not inclusive).

- Use `datetime.timedelta.days` to get the days between `start` and `end`.
- Use `int()` to convert the result to an integer and `range()` to iterate over each day.
- Use a list comprehension and `datetime.timedelta()` to create a list of `datetime.date` objects.

```
1 from datetime import timedelta, date
2
3 def daterange(start, end):
4     return [start + timedelta(n) for n in range(int((end - start).days))]
```

```
1 from datetime import date
2
3 daterange(date(2020, 10, 1), date(2020, 10, 5))
4 # [date(2020, 10, 1), date(2020, 10, 2), date(2020, 10, 3), date(2020, 10, 4)]
```

Calculates the date of `n` days ago from today.

- Use `datetime.date.today()` to get the current day.
- Use `datetime.timedelta` to subtract `n` days from today's date.

```
1 from datetime import timedelta, date
2
3 def days_ago(n):
4     return date.today() - timedelta(n)
```

```
days_ago(5) # date(2020, 10, 23)
```

Calculates the day difference between two dates.

- Subtract `start` from `end` and use `datetime.timedelta.days` to get the day difference.

```
1 def days_diff(start, end):
2     return (end - start).days
```

```
1 from datetime import date
2
3 days_diff(date(2020, 10, 25), date(2020, 10, 28)) # 3
```

Calculates the date of `n` days from today.

- Use `datetime.date.today()` to get the current day.
- Use `datetime.timedelta` to add `n` days from today's date.

```
1 from datetime import timedelta, date
2
3 def days_from_now(n):
4     return date.today() + timedelta(n)
```

```
days_from_now(5) # date(2020, 11, 02)
```

Decapitalizes the first letter of a string.

- Use list slicing and `str.lower()` to decapitalize the first letter of the string.
- Use `str.join()` to combine the lowercase first letter with the rest of the characters.
- Omit the `upper_rest` parameter to keep the rest of the string intact, or set it to `True` to convert to uppercase.

```
1 def decapitalize(s, upper_rest = False):
2     return ''.join([s[:1].lower(), (s[1:]).upper() if upper_rest else s[1:]])
```

```
1 decapitalize('FooBar') # 'fooBar'
2 decapitalize('FooBar', True) # 'FOOBAR'
```

Deep flattens a list.

- Use recursion.
- Use `isinstance()` with `collections.abc.Iterable` to check if an element is iterable.
- If it is iterable, apply `deep_flatten()` recursively, otherwise return `[lst]`.

```
1 from collections.abc import Iterable
2
3 def deep_flatten(lst):
4     return ([a for i in lst for a in
5             deep_flatten(i)] if isinstance(lst, Iterable) else [lst])
```

```
deep_flatten([1, [2], [[3], 4], 5]) # [1, 2, 3, 4, 5]
```

Converts an angle from degrees to radians.

- Use `math.pi` and the degrees to radians formula to convert the angle from degrees to radians.

```
1 from math import pi
2
3 def degrees_to_rads(deg):
```

```
4     return (deg * pi) / 180.0
```

```
degrees_to_rads(180) # ~3.1416
```

Invokes the provided function after `ms` milliseconds.

- Use `time.sleep()` to delay the execution of `fn` by `ms / 1000` seconds.

```
1 from time import sleep
2
3 def delay(fn, ms, *args):
4     sleep(ms / 1000)
5     return fn(*args)
```

```
delay(lambda x: print(x), 1000, 'later') # prints 'later' after one second
```

Converts a dictionary to a list of tuples.

- Use `dict.items()` and `list()` to get a list of tuples from the given dictionary.

```
1 def dict_to_list(d):
2     return list(d.items())
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 dict_to_list(d)
3 # [('one', 1), ('three', 3), ('five', 5), ('two', 2), ('four', 4)]
```

Calculates the difference between two iterables, without filtering duplicate values.

- Create a `set` from `b`.
- Use a list comprehension on `a` to only keep values not contained in the previously created set, `_b`.

```
1 def difference(a, b):
2     _b = set(b)
3     return [item for item in a if item not in _b]
```

```
difference([1, 2, 3], [1, 2, 4]) # [3]
```

Returns the difference between two lists, after applying the provided function to each list element of both.

- Create a `set`, using `map()` to apply `fn` to each element in `b`.
- Use a list comprehension in combination with `fn` on `a` to only keep values not contained in the previously created set, `_b`.

```
1 def difference_by(a, b, fn):
2     _b = set(map(fn, b))
3     return [item for item in a if fn(item) not in _b]
```

```
1 from math import floor
2
3 difference_by([2.1, 1.2], [2.3, 3.4], floor) # [1.2]
4 difference_by([{ 'x': 2 }, { 'x': 1 }], [{ 'x': 1 }], lambda v : v['x'])
5 # [ { x: 2 } ]
```

Converts a number to a list of digits.

- Use `map()` combined with `int` on the string representation of `n` and return a list from the result.

```
1 def digitize(n):
2     return list(map(int, str(n)))
```

```
digitize(123) # [1, 2, 3]
```

Returns a list with `n` elements removed from the left.

- Use slice notation to remove the specified number of elements from the left.
- Omit the last argument, `n`, to use a default value of `1`.

```
1 def drop(a, n = 1):
2     return a[n:]
```

```
1 drop([1, 2, 3]) # [2, 3]
2 drop([1, 2, 3], 2) # [3]
3 drop([1, 2, 3], 42) # []
```

Returns a list with `n` elements removed from the right.

- Use slice notation to remove the specified number of elements from the right.
- Omit the last argument, `n`, to use a default value of `1`.

```
1 def drop_right(a, n = 1):
2     return a[:~n]
```

```
1 drop_right([1, 2, 3]) # [1, 2]
2 drop_right([1, 2, 3], 2) # [1]
3 drop_right([1, 2, 3], 42) # []
```

Checks if the provided function returns `True` for every element in the list.

- Use `all()` in combination with `map()` and `fn` to check if `fn` returns `True` for all elements in the list.

```
1 def every(lst, fn = lambda x: x):
2     return all(map(fn, lst))
```

```
1 every([4, 2, 3], lambda x: x > 1) # True
2 every([1, 2, 3]) # True
```

Returns every `nth` element in a list.

- Use slice notation to create a new list that contains every `nth` element of the given list.

```
1 def every_nth(lst, nth):
2     return lst[nth - 1::nth]
```

```
every_nth([1, 2, 3, 4, 5, 6], 2) # [ 2, 4, 6 ]
```

Calculates the factorial of a number.

- Use recursion.
- If `num` is less than or equal to `1`, return `1`.
- Otherwise, return the product of `num` and the factorial of `num - 1`.
- Throws an exception if `num` is a negative or a floating point number.

```
1 def factorial(num):
2     if not ((num >= 0) and (num % 1 == 0)):
3         raise Exception("Number can't be floating point or negative.")
4     return 1 if num == 0 else num * factorial(num - 1)
```

```
factorial(6) # 720
```

unlisted: true

Converts Fahrenheit to Celsius.

- Follow the conversion formula $C = (F - 32) \times 5/9$.

```
1 def fahrenheit_to_celsius(degrees):
2     return ((degrees - 32) * 5/9)
```

```
fahrenheit_to_celsius(77) # 25.0
```

Generates a list, containing the Fibonacci sequence, up until the nth term.

- Starting with `0` and `1`, use `list.append()` to add the sum of the last two numbers of the list to the end of the list, until the length of the list reaches `n`.
- If `n` is less or equal to `0`, return a list containing `0`.

```
1 def fibonacci(n):
2     if n <= 0:
3         return [0]
4     sequence = [0, 1]
5     while len(sequence) <= n:
6         next_value = sequence[len(sequence) - 1] + sequence[len(sequence) - 2]
7         sequence.append(next_value)
8     return sequence
```

```
fibonacci(7) # [0, 1, 1, 2, 3, 5, 8, 13]
```

Creates a list with the non-unique values filtered out.

- Use `collections.Counter` to get the count of each value in the list.
- Use a list comprehension to create a list containing only the unique values.

```
1 from collections import Counter
2
3 def filter_non_unique(lst):
4     return [item for item, count in Counter(lst).items() if count == 1]
```

```
filter_non_unique([1, 2, 2, 3, 4, 4, 5]) # [1, 3, 5]
```

Creates a list with the unique values filtered out.

- Use `collections.Counter` to get the count of each value in the list.
- Use a list comprehension to create a list containing only the non-unique values.

```
1 from collections import Counter
2
3 def filter_unique(lst):
4     return [item for item, count in Counter(lst).items() if count > 1]
```

```
filter_unique([1, 2, 2, 3, 4, 4, 5]) # [2, 4]
```

Finds the value of the first element in the given list that satisfies the provided testing function.

- Use a list comprehension and `next()` to return the first element in `lst` for which `fn` returns `True`.

```
1 def find(lst, fn):
2     return next(x for x in lst if fn(x))
```

```
find([1, 2, 3, 4], lambda n: n % 2 == 1) # 1
```

Finds the index of the first element in the given list that satisfies the provided testing function.

- Use a list comprehension, `enumerate()` and `next()` to return the index of the first element in `lst` for which `fn` returns `True`.

```
1 def find_index(lst, fn):
2     return next(i for i, x in enumerate(lst) if fn(x))
```

```
find_index([1, 2, 3, 4], lambda n: n % 2 == 1) # 0
```

Finds the indexes of all elements in the given list that satisfy the provided testing function.

- Use `enumerate()` and a list comprehension to return the indexes of the all element in `lst` for which `fn` returns `True`.

```
1 def find_index_of_all(lst, fn):
2     return [i for i, x in enumerate(lst) if fn(x)]
```

```
find_index_of_all([1, 2, 3, 4], lambda n: n % 2 == 1) # [0, 2]
```

Finds the first key in the provided dictionary that has the given value.

- Use `dictionary.items()` and `next()` to return the first key that has a value equal to `val`.

```
1 def find_key(dict, val):
2     return next(key for key, value in dict.items() if value == val)
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 find_key(ages, 11) # 'Isabel'
```

Finds all keys in the provided dictionary that have the given value.

- Use `dictionary.items()`, a generator and `list()` to return all keys that have a value equal to `val`.

```
1 def find_keys(dict, val):
2     return list(key for key, value in dict.items() if value == val)
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 10,
5 }
6 find_keys(ages, 10) # [ 'Peter', 'Anna' ]
```

Finds the value of the last element in the given list that satisfies the provided testing function.

- Use a list comprehension and `next()` to return the last element in `lst` for which `fn` returns `True`.

```
1 def find_last(lst, fn):
2     return next(x for x in lst[::-1] if fn(x))
```

```
find_last([1, 2, 3, 4], lambda n: n % 2 == 1) # 3
```

Finds the index of the last element in the given list that satisfies the provided testing function.

- Use a list comprehension, `enumerate()` and `next()` to return the index of the last element in `lst` for which `fn` returns `True`.

```
1 def find_last_index(lst, fn):
2     return len(lst) - 1 - next(i for i, x in enumerate(lst[::-1]) if fn(x))
```

```
find_last_index([1, 2, 3, 4], lambda n: n % 2 == 1) # 2
```

Finds the items that are parity outliers in a given list.

- Use `collections.Counter` with a list comprehension to count even and odd values in the list.
- Use `collections.Counter.most_common()` to get the most common parity.
- Use a list comprehension to find all elements that do not match the most common parity.

```
1 from collections import Counter
2
3 def find_parity_outliers(nums):
4     return [
5         x for x in nums
6         if x % 2 != Counter([n % 2 for n in nums]).most_common()[0][0]
7     ]
```

```
find_parity_outliers([1, 2, 3, 4, 6]) # [1, 3]
```

Flattens a list of lists once.

- Use a list comprehension to extract each value from sub-lists in order.

```
1 def flatten(lst):
2     return [x for y in lst for x in y]
```

```
flatten([[1, 2, 3, 4], [5, 6, 7, 8]]) # [1, 2, 3, 4, 5, 6, 7, 8]
```

Executes the provided function once for each list element.

- Use a `for` loop to execute `fn` for each element in `itr`.

```
1 def for_each(itr, fn):
2     for el in itr:
3         fn(el)
```

```
for_each([1, 2, 3], print) # 1 2 3
```

Executes the provided function once for each list element, starting from the list's last element.

- Use a `for` loop in combination with slice notation to execute `fn` for each element in `itr`, starting from the last one.

```
1 def for_each_right(itr, fn):
2     for el in itr[::-1]:
3         fn(el)
```

```
for_each_right([1, 2, 3], print) # 3 2 1
```

Creates a dictionary with the unique values of a list as keys and their frequencies as the values.

- Use `collections.defaultdict()` to store the frequencies of each unique element.
- Use `dict()` to return a dictionary with the unique elements of the list as keys and their frequencies as the values.

```
1 from collections import defaultdict
2
3 def frequencies(lst):
4     freq = defaultdict(int)
5     for val in lst:
6         freq[val] += 1
7     return dict(freq)
```

```
frequencies(['a', 'b', 'a', 'c', 'a', 'a', 'b']) # { 'a': 4, 'b': 2, 'c': 1 }
```

Converts a date from its ISO-8601 representation.

- Use `datetime.datetime.fromisoformat()` to convert the given ISO-8601 date to a `datetime.datetime` object.

```
1 from datetime import datetime
2
3 def from_iso_date(d):
4     return datetime.fromisoformat(d)
```

```
from_iso_date('2020-10-28T12:30:59.000000') # 2020-10-28 12:30:59
```

Calculates the greatest common divisor of a list of numbers.

- Use `functools.reduce()` and `math.gcd()` over the given list.

```
1 from functools import reduce
2 from math import gcd as _gcd
3
4 def gcd(numbers):
5     return reduce(_gcd, numbers)
```

```
gcd([8, 36, 28]) # 4
```

Initializes a list containing the numbers in the specified range where `start` and `end` are inclusive and the ratio between two terms is `step`.

Returns an error if `step` equals `1`.

- Use `range()`, `math.log()` and `math.floor()` and a list comprehension to create a list of the appropriate length, applying the step for each element.
- Omit the second argument, `start`, to use a default value of `1`.
- Omit the third argument, `step`, to use a default value of `2`.

```
1 from math import floor, log
2
3 def geometric_progression(end, start=1, step=2):
4     return [start * step ** i for i in range(floor(log(end / start)
5                                                 / log(step)) + 1)]
```

```
1 geometric_progression(256) # [1, 2, 4, 8, 16, 32, 64, 128, 256]
2 geometric_progression(256, 3) # [3, 6, 12, 24, 48, 96, 192]
3 geometric_progression(256, 1, 4) # [1, 4, 16, 64, 256]
```

Retrieves the value of the nested key indicated by the given selector list from a dictionary or list.

- Use `functools.reduce()` to iterate over the `selectors` list.

- Apply `operatorgetitem()` for each key in `selectors`, retrieving the value to be used as the iteratee for the next iteration.

```

1 from functools import reduce
2 from operator import getitem
3
4 def get(d, selectors):
5     return reduce(getitem, selectors, d)

```

```

1 users = {
2     'freddy': {
3         'name': {
4             'first': 'fred',
5             'last': 'smith'
6         },
7         'postIds': [1, 2, 3]
8     }
9 }
10 get(users, ['freddy', 'name', 'last']) # 'smith'
11 get(users, ['freddy', 'postIds', 1]) # 2

```

Groups the elements of a list based on the given function.

- Use `collections.defaultdict` to initialize a dictionary.
- Use `fn` in combination with a `for` loop and `dict.append()` to populate the dictionary.
- Use `dict()` to convert it to a regular dictionary.

```

1 from collections import defaultdict
2
3 def group_by(lst, fn):
4     d = defaultdict(list)
5     for el in lst:
6         d[fn(el)].append(el)
7     return dict(d)

```

```

1 from math import floor
2
3 group_by([6.1, 4.2, 6.3], floor) # {4: [4.2], 6: [6.1, 6.3]}
4 group_by(['one', 'two', 'three'], len) # {3: ['one', 'two'], 5: ['three']}

```

Calculates the Hamming distance between two values.

- Use the XOR operator (`^`) to find the bit difference between the two numbers.
- Use `bin()` to convert the result to a binary string.
- Convert the string to a list and use `count()` of `str` class to count and return the number of `1`s in it.

```

1 def hamming_distance(a, b):
2     return bin(a ^ b).count('1')

```

```
hamming_distance(2, 3) # 1
```

Checks if there are duplicate values in a flat list.

- Use `set()` on the given list to remove duplicates, compare its length with the length of the list.

```
1 def has_duplicates(lst):
2     return len(lst) != len(set(lst))
```

```
1 x = [1, 2, 3, 4, 5, 5]
2 y = [1, 2, 3, 4, 5]
3 has_duplicates(x) # True
4 has_duplicates(y) # False
```

Checks if two lists contain the same elements regardless of order.

- Use `set()` on the combination of both lists to find the unique values.
- Iterate over them with a `for` loop comparing the `count()` of each unique value in each list.
- Return `False` if the counts do not match for any element, `True` otherwise.

```
1 def have_same_contents(a, b):
2     for v in set(a + b):
3         if a.count(v) != b.count(v):
4             return False
5     return True
```

```
have_same_contents([1, 2, 4], [2, 4, 1]) # True
```

Returns the head of a list.

- Use `lst[0]` to return the first element of the passed list.

```
1 def head(lst):
2     return lst[0]
```

```
head([1, 2, 3]) # 1
```

Converts a hexadecimal color code to a tuple of integers corresponding to its RGB components.

- Use a list comprehension in combination with `int()` and list slice notation to get the RGB components from the hexadecimal string.
- Use `tuple()` to convert the resulting list to a tuple.

```
1 def hex_to_rgb(hex):
2     return tuple(int(hex[i:i+2], 16) for i in (0, 2, 4))
```

```
hex_to_rgb('FFA501') # (255, 165, 1)
```

Checks if the given number falls within the given range.

- Use arithmetic comparison to check if the given number is in the specified range.
- If the second parameter, `end`, is not specified, the range is considered to be from `0` to `start`.

```
1 def in_range(n, start, end = 0):
2     return start <= n <= end if end >= start else end <= n <= start
```

```
1 in_range(3, 2, 5) # True
2 in_range(3, 4) # True
3 in_range(2, 3, 5) # False
4 in_range(3, 2) # False
```

Checks if all the elements in `values` are included in `lst`.

- Check if every value in `values` is contained in `lst` using a `for` loop.
- Return `False` if any one value is not found, `True` otherwise.

```
1 def includes_all(lst, values):
2     for v in values:
3         if v not in lst:
4             return False
5     return True
```

```
1 includes_all([1, 2, 3, 4], [1, 4]) # True
2 includes_all([1, 2, 3, 4], [1, 5]) # False
```

Checks if any element in `values` is included in `lst`.

- Check if any value in `values` is contained in `lst` using a `for` loop.
- Return `True` if any one value is found, `False` otherwise.

```
1 def includes_any(lst, values):
2     for v in values:
3         if v in lst:
4             return True
5     return False
```

```
1 includes_any([1, 2, 3, 4], [2, 9]) # True
2 includes_any([1, 2, 3, 4], [8, 9]) # False
```

Returns a list of indexes of all the occurrences of an element in a list.

- Use `enumerate()` and a list comprehension to check each element for equality with `value` and adding `i` to the result.

```
1 def index_of_all(lst, value):
2     return [i for i, x in enumerate(lst) if x == value]
```

```
1 index_of_all([1, 2, 1, 4, 5, 1], 1) # [0, 2, 5]
2 index_of_all([1, 2, 3, 4], 6) # []
```

Returns all the elements of a list except the last one.

- Use `lst[:-1]` to return all but the last element of the list.

```
1 def initial(lst):
2     return lst[:-1]
```

```
initial([1, 2, 3]) # [1, 2]
```

Initializes a 2D list of given width and height and value.

- Use a list comprehension and `range()` to generate `h` rows where each is a list with length `h`, initialized with `val`.
- Omit the last argument, `val`, to set the default value to `None`.

```
1 def initialize_2d_list(w, h, val = None):
2     return [[val for x in range(w)] for y in range(h)]
```

```
initialize_2d_list(2, 2, 0) # [[0, 0], [0, 0]]
```

Initializes a list containing the numbers in the specified range where `start` and `end` are inclusive with their common difference `step`.

- Use `list()` and `range()` to generate a list of the appropriate length, filled with the desired values in the given range.
- Omit `start` to use the default value of `0`.
- Omit `step` to use the default value of `1`.

```
1 def initialize_list_with_range(end, start = 0, step = 1):
2     return list(range(start, end + 1, step))
```

```
1 initialize_list_with_range(5) # [0, 1, 2, 3, 4, 5]
2 initialize_list_with_range(7, 3) # [3, 4, 5, 6, 7]
3 initialize_list_with_range(9, 0, 2) # [0, 2, 4, 6, 8]
```

Initializes and fills a list with the specified value.

- Use a list comprehension and `range()` to generate a list of length equal to `n`, filled with the desired values.
- Omit `val` to use the default value of `0`.

```
1 def initialize_list_with_values(n, val = 0):
2     return [val for x in range(n)]
```

```
initialize_list_with_values(5, 2) # [2, 2, 2, 2]
```

Returns a list of elements that exist in both lists.

- Create a `set` from `a` and `b`.
- Use the built-in set operator `&` to only keep values contained in both sets, then transform the `set` back into a `list`.

```
1 def intersection(a, b):
2     _a, _b = set(a), set(b)
3     return list(_a & _b)
```

```
intersection([1, 2, 3], [4, 3, 2]) # [2, 3]
```

Returns a list of elements that exist in both lists, after applying the provided function to each list element of both.

- Create a `set`, using `map()` to apply `fn` to each element in `b`.
- Use a list comprehension in combination with `fn` on `a` to only keep values contained in both lists.

```
1 def intersection_by(a, b, fn):
2     _b = set(map(fn, b))
3     return [item for item in a if fn(item) in _b]
```

```
1 from math import floor
2
3 intersection_by([2.1, 1.2], [2.3, 3.4], floor) # [2.1]
```

Inverts a dictionary with unique hashable values.

- Use `dictionary.items()` in combination with a list comprehension to create a new dictionary with the values and keys inverted.

```
1 def invert_dictionary(obj):
2     return { value: key for key, value in obj.items() }
```

```
1 ages = {
```

```
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5   }
6 invert_dictionary(ages) # { 10: 'Peter', 11: 'Isabel', 9: 'Anna' }
```

Checks if a string is an anagram of another string (case-insensitive, ignores spaces, punctuation and special characters).

- Use `str.isalnum()` to filter out non-alphanumeric characters, `str.lower()` to transform each character to lowercase.
- Use `collections.Counter` to count the resulting characters for each string and compare the results.

```
1 from collections import Counter
2
3 def is_anagram(s1, s2):
4     return Counter(
5         c.lower() for c in s1 if c.isalnum()
6     ) == Counter(
7         c.lower() for c in s2 if c.isalnum()
8     )
```

```
is_anagram('#anagram', 'Nag a ram!') # True
```

Checks if the elements of the first list are contained in the second one regardless of order.

- Use `count()` to check if any value in `a` has more occurrences than it has in `b`.
- Return `False` if any such value is found, `True` otherwise.

```
1 def is_contained_in(a, b):
2     for v in set(a):
3         if a.count(v) > b.count(v):
4             return False
5     return True
```

```
is_contained_in([1, 4], [2, 4, 1]) # True
```

unlisted: true

Checks if the first numeric argument is divisible by the second one.

- Use the modulo operator (`%`) to check if the remainder is equal to `0`.

```
1 def is_divisible(dividend, divisor):
2     return dividend % divisor == 0
```

```
is_divisible(6, 3) # True
```

unlisted: true

Checks if the given number is even.

- Check whether a number is odd or even using the modulo (`%`) operator.
- Return `True` if the number is even, `False` if the number is odd.

```
1 def is_even(num):  
2     return num % 2 == 0
```

```
is_even(3) # False
```

unlisted: true

Checks if the given number is odd.

- Checks whether a number is even or odd using the modulo (`%`) operator.
- Returns `True` if the number is odd, `False` if the number is even.

```
1 def is_odd(num):  
2     return num % 2 != 0
```

```
is_odd(3) # True
```

Checks if the provided integer is a prime number.

- Return `False` if the number is `0`, `1`, a negative number or a multiple of `2`.
- Use `all()` and `range()` to check numbers from `3` to the square root of the given number.
- Return `True` if none divides the given number, `False` otherwise.

```
1 from math import sqrt  
2  
3 def is_prime(n):  
4     if n <= 1 or (n % 2 == 0 and n > 2):  
5         return False  
6     return all(n % i for i in range(3, int(sqrt(n)) + 1, 2))
```

```
is_prime(11) # True
```

Checks if the given date is a weekday.

- Use `datetime.datetime.weekday()` to get the day of the week as an integer.
- Check if the day of the week is less than or equal to `4`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime
2
3 def is_weekday(d = datetime.today()):
4     return d.weekday() <= 4
```

```
1 from datetime import date
2
3 is_weekday(date(2020, 10, 25)) # False
4 is_weekday(date(2020, 10, 28)) # True
```

Checks if the given date is a weekend.

- Use `datetime.datetime.weekday()` to get the day of the week as an integer.
- Check if the day of the week is greater than `4`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime
2
3 def is_weekend(d = datetime.today()):
4     return d.weekday() > 4
```

```
1 from datetime import date
2
3 is_weekend(date(2020, 10, 25)) # True
4 is_weekend(date(2020, 10, 28)) # False
```

Converts a string to kebab case.

- Use `re.sub()` to replace any `-` or `_` with a space, using the regexp `r"(_|-)+"`.
- Use `re.sub()` to match all words in the string, `str.lower()` to lowercase them.
- Finally, use `str.join()` to combine all word using `-` as the separator.

```
1 from re import sub
2
3 def kebab(s):
4     return '-'.join(
5         sub(r"(\s|_|-)+", " ",
6             sub(r"[A-Z]{2,}(?=[A-Z][a-z]+[0-9]*|\b)|[A-Z]?[a-z]+[0-9]*|[A-Z]|[0-9]+",
7                 lambda mo: ' ' + mo.group(0).lower(), s)).split())
```

```
1 kebab('camelCase') # 'camel-case'
2 kebab('some text') # 'some-text'
3 kebab('some-mixed_string With spaces_underscores-and-hyphens')
4 # 'some-mixed-string-with-spaces-underscores-and-hyphens'
5 kebab('AllThe-small Things') # 'all-the-small-things'
```

Checks if the given key exists in a dictionary.

- Use the `in` operator to check if `d` contains `key`.

```
1 def key_in_dict(d, key):  
2     return (key in d)
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}  
2 key_in_dict(d, 'three') # True
```

Finds the key of the maximum value in a dictionary.

- Use `max()` with the `key` parameter set to `dict.get()` to find and return the key of the maximum value in the given dictionary.

```
1 def key_of_max(d):  
2     return max(d, key = d.get)
```

```
key_of_max({'a':4, 'b':0, 'c':13}) # c
```

Finds the key of the minimum value in a dictionary.

- Use `min()` with the `key` parameter set to `dict.get()` to find and return the key of the minimum value in the given dictionary.

```
1 def key_of_min(d):  
2     return min(d, key = d.get)
```

```
key_of_min({'a':4, 'b':0, 'c':13}) # b
```

Creates a flat list of all the keys in a flat dictionary.

- Use `dict.keys()` to return the keys in the given dictionary.
- Return a `list()` of the previous result.

```
1 def keys_only(flat_dict):  
2     return list(flat_dict.keys())
```

```
1 ages = {  
2     'Peter': 10,  
3     'Isabel': 11,  
4     'Anna': 9,  
5 }  
6 keys_only(ages) # ['Peter', 'Isabel', 'Anna']
```

unlisted: true

Converts kilometers to miles.

- Follows the conversion formula `mi = km * 0.621371`.

```
1 def km_to_miles(km):
2     return km * 0.621371
```

```
km_to_miles(8.1) # 5.0331051
```

Returns the last element in a list.

- Use `lst[-1]` to return the last element of the passed list.

```
1 def last(lst):
2     return lst[-1]
```

```
last([1, 2, 3]) # 3
```

Returns the least common multiple of a list of numbers.

- Use `functools.reduce()`, `math.gcd()` and `lcm(x,y) = x * y / gcd(x,y)` over the given list.

```
1 from functools import reduce
2 from math import gcd
3
4 def lcm(numbers):
5     return reduce((lambda x, y: int(x * y / gcd(x, y))), numbers)
```

```
1 lcm([12, 7]) # 84
2 lcm([1, 3, 4, 5]) # 60
```

Takes any number of iterable objects or objects with a length property and returns the longest one.

- Use `max()` with `len()` as the `key` to return the item with the greatest length.
- If multiple objects have the same length, the first one will be returned.

```
1 def longest_item(*args):
2     return max(args, key = len)
```

```
1 longest_item('this', 'is', 'a', 'testcase') # 'testcase'
2 longest_item([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]) # [1, 2, 3, 4, 5]
```

```
3 longest_item([1, 2, 3], 'foobar') # 'foobar'
```

Maps the values of a list to a dictionary using a function, where the key-value pairs consist of the original value as the key and the result of the function as the value.

- Use `map()` to apply `fn` to each value of the list.
- Use `zip()` to pair original values to the values produced by `fn`.
- Use `dict()` to return an appropriate dictionary.

```
1 def map_dictionary(itr, fn):
2     return dict(zip(itr, map(fn, itr)))
```

```
map_dictionary([1, 2, 3], lambda x: x * x) # { 1: 1, 2: 4, 3: 9 }
```

Creates a dictionary with the same keys as the provided dictionary and values generated by running the provided function for each value.

- Use `dict.items()` to iterate over the dictionary, assigning the values produced by `fn` to each key of a new dictionary.

```
1 def map_values(obj, fn):
2     return dict((k, fn(v)) for k, v in obj.items())
```

```
1 users = {
2     'fred': { 'user': 'fred', 'age': 40 },
3     'pebbles': { 'user': 'pebbles', 'age': 1 }
4 }
5 map_values(users, lambda u : u['age']) # {'fred': 40, 'pebbles': 1}
```

Returns the maximum value of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `max()` to return the maximum value.

```
1 def max_by(lst, fn):
2     return max(map(fn, lst))
```

```
max_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 8
```

Returns the index of the element with the maximum value in a list.

- Use `max()` and `list.index()` to get the maximum value in the list and return its index.

```
1 def max_element_index(arr):
```

```
2     return arr.index(max(arr))
```

```
max_element_index([5, 8, 9, 7, 10, 3, 0]) # 4
```

Returns the `n` maximum elements from the provided list.

- Use `sorted()` to sort the list.
- Use slice notation to get the specified number of elements.
- Omit the second argument, `n`, to get a one-element list.
- If `n` is greater than or equal to the provided list's length, then return the original list (sorted in descending order).

```
1 def max_n(lst, n = 1):
2     return sorted(lst, reverse = True)[:n]
```

```
1 max_n([1, 2, 3]) # [3]
2 max_n([1, 2, 3], 2) # [3, 2]
```

Finds the median of a list of numbers.

- Sort the numbers of the list using `list.sort()`.
- Find the median, which is either the middle element of the list if the list length is odd or the average of the two middle elements if the list length is even.
- `statistics.median()` provides similar functionality to this snippet.

```
1 def median(list):
2     list.sort()
3     list_length = len(list)
4     if list_length % 2 == 0:
5         return (list[int(list_length / 2) - 1] + list[int(list_length / 2)]) / 2
6     return float(list[int(list_length / 2)])
```

```
1 median([1, 2, 3]) # 2.0
2 median([1, 2, 3, 4]) # 2.5
```

Merges two or more lists into a list of lists, combining elements from each of the input lists based on their positions.

- Use `max()` combined with a list comprehension to get the length of the longest list in the arguments.
- Use `range()` in combination with the `max_length` variable to loop as many times as there are elements in the longest list.
- If a list is shorter than `max_length`, use `fill_value` for the remaining items (defaults to `None`).
- `zip()` and `itertools.zip_longest()` provide similar functionality to this snippet.

```
1 def merge(*args, fill_value = None):
2     max_length = max([len(lst) for lst in args])
3     result = []
4     for i in range(max_length):
```

```
5     result.append([
6         args[k][i] if i < len(args[k]) else fill_value for k in range(len(args))
7     ])
8     return result
```

```
1 merge([('a', 'b'), [1, 2], [True, False])] # [['a', 1, True], ['b', 2, False]]
2 merge([('a'), [1, 2], [True, False])] # [['a', 1, True], [None, 2, False]]
3 merge([('a'), [1, 2], [True, False], fill_value = '_')]
4 # [['a', 1, True], ['_', 2, False]]
```

Merges two or more dictionaries.

- Create a new `dict` and loop over `dicts`, using `dictionary.update()` to add the key-value pairs from each one to the result.

```
1 def merge_dictionaries(*dicts):
2     res = dict()
3     for d in dicts:
4         res.update(d)
5     return res
```

```
1 ages_one = {
2     'Peter': 10,
3     'Isabel': 11,
4 }
5 ages_two = {
6     'Anna': 9
7 }
8 merge_dictionaries(ages_one, ages_two)
9 # { 'Peter': 10, 'Isabel': 11, 'Anna': 9 }
```

unlisted: true

Converts miles to kilometers.

- Follows the conversion formula `km = mi * 1.609344`.

```
1 def miles_to_km(miles):
2     return miles * 1.609344
```

```
miles_to_km(5.03) # 8.09500032
```

Returns the minimum value of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `min()` to return the minimum value.

```
1 def min_by(lst, fn):
2     return min(map(fn, lst))
```

```
min_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 2
```

Returns the index of the element with the minimum value in a list.

- Use `min()` and `list.index()` to obtain the minimum value in the list and then return its index.

```
1 def min_element_index(arr):
2     return arr.index(min(arr))
```

```
min_element_index([3, 5, 2, 6, 10, 7, 9]) # 2
```

Returns the `n` minimum elements from the provided list.

- Use `sorted()` to sort the list.
- Use slice notation to get the specified number of elements.
- Omit the second argument, `n`, to get a one-element list.
- If `n` is greater than or equal to the provided list's length, then return the original list (sorted in ascending order).

```
1 def min_n(lst, n = 1):
2     return sorted(lst, reverse = False)[:n]
```

```
1 min_n([1, 2, 3]) # [1]
2 min_n([1, 2, 3], 2) # [1, 2]
```

Calculates the month difference between two dates.

- Subtract `start` from `end` and use `datetime.timedelta.days` to get the day difference.
- Divide by `30` and use `math.ceil()` to get the difference in months (rounded up).

```
1 from math import ceil
2
3 def months_diff(start, end):
4     return ceil((end - start).days / 30)
```

```
1 from datetime import date
2
3 months_diff(date(2020, 10, 28), date(2020, 11, 25)) # 1
```

Returns the most frequent element in a list.

- Use `set()` to get the unique values in `lst`.

- Use `max()` to find the element that has the most appearances.

```
1 def most_frequent(lst):
2     return max(set(lst), key = lst.count)
```

```
most_frequent([1, 2, 1, 2, 3, 2, 1, 4, 2]) #2
```

Generates a string with the given string value repeated `n` number of times.

- Repeat the string `n` times, using the `*` operator.

```
1 def n_times_string(s, n):
2     return (s * n)
```

```
n_times_string('py', 4) #'pypypy'
```

Checks if the provided function returns `True` for at least one element in the list.

- Use `all()` and `fn` to check if `fn` returns `False` for all the elements in the list.

```
1 def none(lst, fn = lambda x: x):
2     return all(not fn(x) for x in lst)
```

```
1 none([0, 1, 2, 0], lambda x: x >= 2) # False
2 none([0, 0, 0]) # True
```

Maps a number from one range to another range.

- Return `num` mapped between `outMin - outMax` from `inMin - inMax`.

```
1 def num_to_range(num, inMin, inMax, outMin, outMax):
2     return outMin + (float(num - inMin) / float(inMax - inMin)) * (outMax
3                           - outMin))
```

```
num_to_range(5, 0, 10, 0, 100) # 50.0
```

Moves the specified amount of elements to the end of the list.

- Use slice notation to get the two slices of the list and combine them before returning.

```
1 def offset(lst, offset):
2     return lst[offset:] + lst[:offset]
```

```
1 offset([1, 2, 3, 4, 5], 2) # [3, 4, 5, 1, 2]
2 offset([1, 2, 3, 4, 5], -2) # [4, 5, 1, 2, 3]
```

Pads a string on both sides with the specified character, if it's shorter than the specified length.

- Use `str.ljust()` and `str.rjust()` to pad both sides of the given string.
- Omit the third argument, `char`, to use the whitespace character as the default padding character.

```
1 from math import floor
2
3 def pad(s, length, char = ' '):
4     return s.rjust(floor((len(s) + length)/2), char).ljust(length, char)
```

```
1 pad('cat', 8) #   cat   '
2 pad('42', 6, '0') # '004200'
3 pad('foobar', 3) # 'foobar'
```

Pads a given number to the specified length.

- Use `str.zfill()` to pad the number to the specified length, after converting it to a string.

```
1 def pad_number(n, l):
2     return str(n).zfill(l)
```

```
pad_number(1234, 6); # '001234'
```

Checks if the given string is a palindrome.

- Use `str.lower()` and `re.sub()` to convert to lowercase and remove non-alphanumeric characters from the given string.
- Then, compare the new string with its reverse, using slice notation.

```
1 from re import sub
2
3 def palindrome(s):
4     s = sub('[\W_]', '', s.lower())
5     return s == s[::-1]
```

```
palindrome('taco cat') # True
```

Converts a list of dictionaries into a list of values corresponding to the specified key .

- Use a list comprehension and `dict.get()` to get the value of `key` for each dictionary in `lst`.

```
1 def pluck(lst, key):
2     return [x.get(key) for x in lst]
```

```
1 simpsons = [
2     { 'name': 'lisa', 'age': 8 },
3     { 'name': 'homer', 'age': 36 },
4     { 'name': 'marge', 'age': 34 },
5     { 'name': 'bart', 'age': 10 }
6 ]
7 pluck(simpsons, 'age') # [8, 36, 34, 10]
```

Returns the powerset of a given iterable.

- Use `list()` to convert the given value to a list.
- Use `range()` and `itertools.combinations()` to create a generator that returns all subsets.
- Use `itertools.chain.from_iterable()` and `list()` to consume the generator and return a list.

```
1 from itertools import chain, combinations
2
3 def powerset(iterable):
4     s = list(iterable)
5     return list(chain.from_iterable(combinations(s, r) for r in range(len(s)+1)))
```

```
powerset([1, 2]) # [(), (1,), (2,), (1, 2)]
```

Converts an angle from radians to degrees.

- Use `math.pi` and the radian to degree formula to convert the angle from radians to degrees.

```
1 from math import pi
2
3 def rads_to_degrees(rad):
4     return (rad * 180.0) / pi
```

```
1 from math import pi
2
3 rads_to_degrees(pi / 2) # 90.0
```

Reverses a list or a string.

- Use slice notation to reverse the list or string.

```
1 def reverse(itr):
2     return itr[::-1]
```

```
1 reverse([1, 2, 3]) # [3, 2, 1]
2 reverse('snippet') # 'teppins'
```

Reverses a number.

- Use `str()` to convert the number to a string, slice notation to reverse it and `str.replace()` to remove the sign.
- Use `float()` to convert the result to a number and `math.copysign()` to copy the original sign.

```
1 from math import copysign
2
3 def reverse_number(n):
4     return copysign(float(str(n)[::-1].replace('-', '')), n)
```

```
1 reverse_number(981) # 189
2 reverse_number(-500) # -5
3 reverse_number(73.6) # 6.37
4 reverse_number(-5.23) # -32.5
```

Converts the values of RGB components to a hexadecimal color code.

- Create a placeholder for a zero-padded hexadecimal value using '`{:02X}`' and copy it three times.
- Use `str.format()` on the resulting string to replace the placeholders with the given values.

```
1 def rgb_to_hex(r, g, b):
2     return ('{:02X}' * 3).format(r, g, b)
```

```
rgb_to_hex(255, 165, 1) # 'FFA501'
```

Moves the specified amount of elements to the start of the list.

- Use slice notation to get the two slices of the list and combine them before returning.

```
1 def roll(lst, offset):
2     return lst[-offset:] + lst[:-offset]
```

```
1 roll([1, 2, 3, 4, 5], 2) # [4, 5, 1, 2, 3]
2 roll([1, 2, 3, 4, 5], -2) # [3, 4, 5, 1, 2]
```

Returns a random element from a list.

- Use `random.choice()` to get a random element from `lst`.

```
1 from random import choice
2
```

```
3 def sample(lst):
4     return choice(lst)
```

```
sample([3, 7, 9, 11]) # 9
```

Randomizes the order of the values of an list, returning a new list.

- Uses the [Fisher-Yates algorithm](#) to reorder the elements of the list.
- `random.shuffle` provides similar functionality to this snippet.

```
1 from copy import deepcopy
2 from random import randint
3
4 def shuffle(lst):
5     temp_lst = deepcopy(lst)
6     m = len(temp_lst)
7     while (m):
8         m -= 1
9         i = randint(0, m)
10        temp_lst[m], temp_lst[i] = temp_lst[i], temp_lst[m]
11    return temp_lst
```

```
1 foo = [1, 2, 3]
2 shuffle(foo) # [2, 3, 1], foo = [1, 2, 3]
```

Returns a list of elements that exist in both lists.

- Use a list comprehension on `a` to only keep values contained in both lists.

```
1 def similarity(a, b):
2     return [item for item in a if item in b]
```

```
similarity([1, 2, 3], [1, 2, 4]) # [1, 2]
```

Converts a string to a URL-friendly slug.

- Use `str.lower()` and `str.strip()` to normalize the input string.
- Use `re.sub()` to replace spaces, dashes and underscores with `-` and remove special characters.

```
1 import re
2
3 def slugify(s):
4     s = s.lower().strip()
5     s = re.sub(r'^[\w\-\_]+', '', s)
6     s = re.sub(r'[\s\-\_]+\+', ' ', s)
7     s = re.sub(r'^\-\+|\-\+$', '', s)
8
9     return s
```

```
slugify('Hello World!') # 'hello-world'
```

Converts a string to snake case.

- Use `re.sub()` to match all words in the string, `str.lower()` to lowercase them.
- Use `re.sub()` to replace any `-` characters with spaces.
- Finally, use `str.join()` to combine all words using `-` as the separator.

```
1 from re import sub
2
3 def snake(s):
4     return '_'.join(
5         sub('([A-Z][a-z]+)', r' \1',
6             sub('([A-Z]+)', r' \1',
7                 s.replace('-', ' '))).split()).lower()
```

```
1 snake('camelCase') # 'camel_case'
2 snake('some text') # 'some_text'
3 snake('some-mixed_string With spaces_underscores-and-hyphens')
4 # 'some_mixed_string_with_spaces_underscores_and_hyphens'
5 snake('AllThe-small Things') # 'all_the_small_things'
```

Checks if the provided function returns `True` for at least one element in the list.

- Use `any()` in combination with `map()` to check if `fn` returns `True` for any element in the list.

```
1 def some(lst, fn = lambda x: x):
2     return any(map(fn, lst))
```

```
1 some([0, 1, 2, 0], lambda x: x >= 2) # True
2 some([0, 0, 1, 0]) # True
```

Sorts one list based on another list containing the desired indexes.

- Use `zip()` and `sorted()` to combine and sort the two lists, based on the values of `indexes`.
- Use a list comprehension to get the first element of each pair from the result.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the third argument.

```
1 def sort_by_indexes(lst, indexes, reverse=False):
2     return [val for (_, val) in sorted(zip(indexes, lst), key=lambda x: \
3                                         x[0], reverse=reverse)]
```

```
1 a = ['eggs', 'bread', 'oranges', 'jam', 'apples', 'milk']
2 b = [3, 2, 6, 4, 1, 5]
3 sort_by_indexes(a, b) # ['apples', 'bread', 'eggs', 'jam', 'milk', 'oranges']
4 sort_by_indexes(a, b, True)
5 # ['oranges', 'milk', 'jam', 'eggs', 'bread', 'apples']
```

Sorts the given dictionary by key.

- Use `dict.items()` to get a list of tuple pairs from `d` and sort it using `sorted()`.
- Use `dict()` to convert the sorted list back to a dictionary.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the second argument.

```
1 def sort_dict_by_key(d, reverse = False):
2     return dict(sorted(d.items(), reverse = reverse))
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 sort_dict_by_key(d) # {'five': 5, 'four': 4, 'one': 1, 'three': 3, 'two': 2}
3 sort_dict_by_key(d, True)
4 # {'two': 2, 'three': 3, 'one': 1, 'four': 4, 'five': 5}
```

Sorts the given dictionary by value.

- Use `dict.items()` to get a list of tuple pairs from `d` and sort it using a lambda function and `sorted()`.
- Use `dict()` to convert the sorted list back to a dictionary.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the second argument.
- **⚠️ NOTICE:** Dictionary values must be of the same type.

```
1 def sort_dict_by_value(d, reverse = False):
2     return dict(sorted(d.items(), key = lambda x: x[1], reverse = reverse))
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 sort_dict_by_value(d) # {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
3 sort_dict_by_value(d, True)
4 # {'five': 5, 'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

Splits a multiline string into a list of lines.

- Use `str.split()` and `'\n'` to match line breaks and create a list.
- `str.splitlines()` provides similar functionality to this snippet.

```
1 def split_lines(s):
2     return s.split('\n')
```

```
1 split_lines('This\nis a\nmultiline\nstring.\n')
2 # ['This', 'is a', 'multiline', 'string.', '']
```

Flattens a list, by spreading its elements into a new list.

- Loop over elements, use `list.extend()` if the element is a list, `list.append()` otherwise.

```
1 def spread(arg):
```

```
2     ret = []
3     for i in arg:
4         ret.extend(i) if isinstance(i, list) else ret.append(i)
5     return ret
```

```
spread([1, 2, 3, [4, 5, 6], [7], 8, 9]) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Calculates the sum of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `sum()` to return the sum of the values.

```
1 def sum_by(lst, fn):
2     return sum(map(fn, lst))
```

```
sum_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 20
```

Returns the sum of the powers of all the numbers from `start` to `end` (both inclusive).

- Use `range()` in combination with a list comprehension to create a list of elements in the desired range raised to the given `power`.
- Use `sum()` to add the values together.
- Omit the second argument, `power`, to use a default power of `2`.
- Omit the third argument, `start`, to use a default starting value of `1`.

```
1 def sum_of_powers(end, power = 2, start = 1):
2     return sum([(i) ** power for i in range(start, end + 1)])
```

```
1 sum_of_powers(10) # 385
2 sum_of_powers(10, 3) # 3025
3 sum_of_powers(10, 3, 5) # 2925
```

Returns the symmetric difference between two iterables, without filtering out duplicate values.

- Create a `set` from each list.
- Use a list comprehension on each of them to only keep values not contained in the previously created set of the other.

```
1 def symmetric_difference(a, b):
2     (_a, _b) = (set(a), set(b))
3     return [item for item in a if item not in _b] + [item for item in b
4             if item not in _a]
```

```
symmetric_difference([1, 2, 3], [1, 2, 4]) # [3, 4]
```

Returns the symmetric difference between two lists, after applying the provided function to each list element of both.

- Create a `set` by applying `fn` to each element in every list.
- Use a list comprehension in combination with `fn` on each of them to only keep values not contained in the previously created set of the other.

```
1 def symmetric_difference_by(a, b, fn):
2     (_a, _b) = (set(map(fn, a)), set(map(fn, b)))
3     return [item for item in a if fn(item) not in _b] + [item
4             for item in b if fn(item) not in _a]
```

```
1 from math import floor
2
3 symmetric_difference_by([2.1, 1.2], [2.3, 3.4], floor) # [1.2, 3.4]
```

Returns all elements in a list except for the first one.

- Use slice notation to return the last element if the list's length is more than `1`.
- Otherwise, return the whole list.

```
1 def tail(lst):
2     return lst[1:] if len(lst) > 1 else lst
```

```
1 tail([1, 2, 3]) # [2, 3]
2 tail([1]) # [1]
```

Returns a list with `n` elements removed from the beginning.

- Use slice notation to create a slice of the list with `n` elements taken from the beginning.

```
1 def take(itr, n = 1):
2     return itr[:n]
```

```
1 take([1, 2, 3], 5) # [1, 2, 3]
2 take([1, 2, 3], 0) # []
```

Returns a list with `n` elements removed from the end.

- Use slice notation to create a slice of the list with `n` elements taken from the end.

```
1 def take_right(itr, n = 1):
2     return itr[-n:]
```

```
1 take_right([1, 2, 3], 2) # [2, 3]
2 take_right([1, 2, 3]) # [3]
```

Returns the binary representation of the given number.

- Use `bin()` to convert a given decimal number into its binary equivalent.

```
1 def to_binary(n):
2     return bin(n)
```

```
to_binary(100) # 0b1100100
```

Combines two lists into a dictionary, where the elements of the first one serve as the keys and the elements of the second one serve as the values.

The values of the first list need to be unique and hashable.

- Use `zip()` in combination with `dict()` to combine the values of the two lists into a dictionary.

```
1 def to_dictionary(keys, values):
2     return dict(zip(keys, values))
```

```
to_dictionary(['a', 'b'], [1, 2]) # { a: 1, b: 2 }
```

Returns the hexadecimal representation of the given number.

- Use `hex()` to convert a given decimal number into its hexadecimal equivalent.

```
1 def to_hex(dec):
2     return hex(dec)
```

```
1 to_hex(41) # 0x29
2 to_hex(332) # 0x14c
```

Converts a date to its ISO-8601 representation.

- Use `datetime.datetime.isoformat()` to convert the given `datetime.datetime` object to an ISO-8601 date.

```
1 from datetime import datetime
2
3 def to_iso_date(d):
4     return d.isoformat()
```

```
1 from datetime import datetime
2
3 to_iso_date(datetime(2020, 10, 25)) # 2020-10-25T00:00:00
```

Converts an integer to its roman numeral representation.

Accepts value between `1` and `3999` (both inclusive).

- Create a lookup list containing tuples in the form of (roman value, integer).
- Use a `for` loop to iterate over the values in `lookup`.
- Use `divmod()` to update `num` with the remainder, adding the roman numeral representation to the result.

```
1 def to_roman_numeral(num):
2     lookup = [
3         (1000, 'M'),
4         (900, 'CM'),
5         (500, 'D'),
6         (400, 'CD'),
7         (100, 'C'),
8         (90, 'XC'),
9         (50, 'L'),
10        (40, 'XL'),
11        (10, 'X'),
12        (9, 'IX'),
13        (5, 'V'),
14        (4, 'IV'),
15        (1, 'I'),
16    ]
17    res = ''
18    for (n, roman) in lookup:
19        (d, num) = divmod(num, n)
20        res += roman * d
21    return res
```

```
1 to_roman_numeral(3) # 'III'
2 to_roman_numeral(11) # 'XI'
3 to_roman_numeral(1998) # 'MCMXCVIII'
```

Transposes a two-dimensional list.

- Use `*lst` to get the provided list as tuples.
- Use `zip()` in combination with `list()` to create the transpose of the given two-dimensional list.

```
1 def transpose(lst):
2     return list(zip(*lst))
```

```
1 transpose([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
2 # [(1, 4, 7, 10), (2, 5, 8, 11), (3, 6, 9, 12)]
```

Builds a list, using an iterator function and an initial seed value.

- The iterator function accepts one argument (`seed`) and must always return a list with two elements ([`value`, `nextSeed`]) or `False` to terminate.
- Use a generator function, `fn_generator`, that uses a `while` loop to call the iterator function and `yield` the `value` until it returns `False`.
- Use a list comprehension to return the list that is produced by the generator, using the iterator function.

```

1 def unfold(fn, seed):
2     def fn_generator(val):
3         while True:
4             val = fn(val[1])
5             if val == False: break
6             yield val[0]
7     return [i for i in fn_generator([None, seed])]
```

```

1 f = lambda n: False if n > 50 else [-n, n + 10]
2 unfold(f, 10) # [-10, -20, -30, -40, -50]
```

Returns every element that exists in any of the two lists once.

- Create a `set` with all values of `a` and `b` and convert to a `list`.

```

1 def union(a, b):
2     return list(set(a + b))
```

```
union([1, 2, 3], [4, 3, 2]) # [1, 2, 3, 4]
```

Returns every element that exists in any of the two lists once, after applying the provided function to each element of both.

- Create a `set` by applying `fn` to each element in `a`.
- Use a list comprehension in combination with `fn` on `b` to only keep values not contained in the previously created set, `_a`.
- Finally, create a `set` from the previous result and `a` and transform it into a `list`

```

1 def union_by(a, b, fn):
2     _a = set(map(fn, a))
3     return list(set(a + [item for item in b if fn(item) not in _a]))
```

```

1 from math import floor
2
3 union_by([2.1], [1.2, 2.3], floor) # [2.1, 1.2]
```

Returns the unique elements in a given list.

- Create a `set` from the list to discard duplicated values, then return a `list` from it.

```
1 def unique_elements(li):
2     return list(set(li))
```

```
unique_elements([1, 2, 2, 3, 4, 3]) # [1, 2, 3, 4]
```

Returns a flat list of all the values in a flat dictionary.

- Use `dict.values()` to return the values in the given dictionary.
- Return a `list()` of the previous result.

```
1 def values_only(flat_dict):
2     return list(flat_dict.values())
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 values_only(ages) # [10, 11, 9]
```

Returns the weighted average of two or more numbers.

- Use `sum()` to sum the products of the numbers by their weight and to sum the weights.
- Use `zip()` and a list comprehension to iterate over the pairs of values and weights.

```
1 def weighted_average(nums, weights):
2     return sum(x * y for x, y in zip(nums, weights)) / sum(weights)
```

```
weighted_average([1, 2, 3], [0.6, 0.2, 0.3]) # 1.72727
```

Tests a value, `x`, against a testing function, conditionally applying a function.

- Check if the value of `predicate(x)` is `True` and if so return `when_true(x)`, otherwise return `x`.

```
1 def when(predicate, when_true):
2     return lambda x: when_true(x) if predicate(x) else x
```

```
1 double_even_numbers = when(lambda x: x % 2 == 0, lambda x : x * 2)
2 double_even_numbers(2) # 4
3 double_even_numbers(1) # 1
```

Converts a given string into a list of words.

- Use `re.findall()` with the supplied `pattern` to find all matching substrings.
- Omit the second argument to use the default regexp, which matches alphanumeric and hyphens.

```
1 import re
2
3 def words(s, pattern = '[a-zA-Z-]+'):
4     return re.findall(pattern, s)
```

```
1 words('I love Python!!') # ['I', 'love', 'Python']
2 words('python, javaScript & coffee') # ['python', 'javaScript', 'coffee']
3 words('build -q --out one-item', r'\b[a-zA-Z-]+\b')
4 # ['build', 'q', 'out', 'one-item']
```

Python-Docs

Docs



[3.9.7 Documentation](https://docs.python.org/3/)

<https://docs.python.org/3/>



[ds-algo \(forked\) - CodeSandbox](https://codesandbox.io/s/ds-algo-forked-60s1b)

<https://codesandbox.io/s/ds-algo-forked-60s1b>

General Docs:

<https://codesandbox.io/s/ds-algo-forked-lfujh?from-embed>

Built In Functions

- `abs()` function
- `bin()` function
- `id()` function
- `map()` function
- `zip()` function
- `filter()` function
- `reduce()` function
- `sorted()` function
- `enumerate()` function
- `reversed()` function
- `range()` function
- `sum()` function
- `max()` function
- `min()` function
- `eval()` function
- `len()` function
- `ord()` function
- `chr()` function
- `any()` function
- `all()` function
- `globals()` function
- `locals()` function

Lists

Dictionaries

 Copy of dictionaries.ipynb

<https://gist.github.com/bgoonz/df0237e949950dfd81add75e2b95f60a>

```
1 import itertools
2
3 # Given an array of coins and an array of quantities for each coin with the
4 # same index, determine how many distinct sums can be made from non-zero
5 # sets of the coins
6
7 # Note: This problem took a little more working-through, with a failed brute-
8 # force attempt that consisted of finding every combination of coins and
9 # adding them, which failed when I needed to consider >50k coins
10 # the overall number of coins was guaranteed to be less than about 1 million,
11 # so the solution appeared to be a form of divide-and-conquer where each
12 # possible sum for each coin was put into a set at that coin's index in the
13 # original coins array, and then the sums were repeatedly combined into an
14 # aggregate set until every coin possible coin value (given by the coins
15 # array) had been added into the set of sums
16
17 # problem considered "hard," asked by Google
18
19
20 def possibleSums(coins, quantity):
21     # sum_map = set()
22     # start with brute force
23     # total_arr = [coins[i] for i, q in enumerate(quantity) for l in range(q)]
24
25     # for i in range(1, len(total_arr)+1):
26     #     combos = itertools.combinations(total_arr, i)
27     #     print(combos)
28     #     for combo in combos:
29     #         sum_map.add(sum(combo))
30
31     # return len(sum_map)
32
33     # faster?
34     comb_indices = [i for i in range(len(coins))]
35     possible_sums = []
36     for i, c in enumerate(coins):
37         this_set = set()
38         for q in range(1, 1 + quantity[i]):
39             this_set.add(c * q)
40         possible_sums.append(this_set)
41     # print(possible_sums)
42
43     while len(possible_sums) > 1:
44         possible_sums[0] = combine_sets(possible_sums[0], possible_sums[1])
45         possible_sums.pop(1)
46
47     return len(possible_sums[0])
48
49
50 def combine_sets(set1, set2):
51     together_set = set()
52     for item1 in set1:
53         for item2 in set2:
54             together_set.add(item1 + item2)
55             together_set.add(item1)
56
57     for item2 in set2:
58         together_set.add(item2)
59     return together_set
60
```

```

1 # need strings[i] = strings[j] for all patterns[i] = patterns[j] to be true -
2 # give false if strings[i] != strings[j] and patterns[i] = patterns[j] or
3 # strings[i] = strings[j] and patterns[j] != patterns[j] - this last condition
4 # threw me for a bit as an edge case! Need to ensure that each string is unique
5 # to each key, not just that each key corresponds to the given string!
6
7 # from a google interview set, apparently
8 def areFollowingPatterns(strings, patterns):
9     pattern_to_string = {}
10    string_to_pattern = {}
11    for i in range(len(patterns)):
12        # first, check condition that strings are equal for patterns[i]=patterns[j]
13        this_pattern = patterns[i]
14        if patterns[i] in pattern_to_string:
15            if strings[i] != pattern_to_string[this_pattern]:
16                return False
17            else:
18                pattern_to_string[this_pattern] = strings[i]
19
20    # now check condition that patterns are equal for strings[i]=strings[j]
21    # if there are more keys than values, then there is not 1:1 correspondence
22    if len(pattern_to_string.keys()) != len(set(pattern_to_string.values())):
23        return False
24
25    return True
26

```

```

1 # gives True if two duplicate numbers in the nums array are within k distance
2 # (inclusive) of one another, measuring by absolute difference in index
3
4 # did relatively well on this one, made a greater-than/less-than flip error on
5 # the conditional for the true case and needed to rewrite my code to remove
6 # keys from the dictionary without editing it while looping over it, but
7 # otherwise went well!
8
9 # problem considered medium difficulty, from Palantir
10
11
12 def containsCloseNums(nums, k):
13     num_dict = {}
14     # setup keys for each number seen, then list their indices
15     for i, item in enumerate(nums):
16         if item in num_dict:
17             num_dict[item].append(i)
18         else:
19             num_dict[item] = [i]
20
21     # remove all nums that are not repeated
22     # first make a set of keys to remove to prevent editing the dictionary size while iterating over it
23     removals = set()
24     for key in num_dict.keys():
25         if len(num_dict[key]) < 2:
26             removals.add(key)
27
28     # now remove each key from the num_dict that has fewer than two values
29     for key in removals:
30         num_dict.pop(key)
31
32     # now check remaining numbers to see if they fall within the desired range
33     for key in num_dict.keys():
34         last_ind = num_dict[key][0]
35         for next_ind in num_dict[key][1:]:
36             if next_ind - last_ind <= k:

```

```
37         return True
38     last_ind = next_ind
39
40     return False
41
```

Classes

1. [Home](#)
- 2.
3. [Blog](#)
- 4.
5. Python Classes and Interfaces

(Sponsors) Get started learning Python with DataCamp's free [Intro to Python tutorial](#). Learn Data Science by completing interactive coding challenges and watching videos by expert instructors. [Start Now!](#)

Python Classes and Interfaces

Updated on Jan 07, 2020

Note:

This is a excerpt from [Effective Python: 90 Specific Ways to Write Better Python, 2nd Edition](#)

As an object-oriented programming language, Python supports a full range of features, such as inheritance, polymorphism, and encapsulation. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies.

Python's classes and inheritance make it easy to express a program's intended behaviors with objects. They allow you to improve and expand functionality over time. They provide flexibility in an environment of changing requirements. Knowing how to use them well enables you to write maintainable code.

Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Python's built-in dictionary type is wonderful for maintaining dynamic internal state over the lifetime of an object. By dynamic, I mean situations in which you need to do bookkeeping for an unexpected set of identifiers. For example, say that I want to record the grades of a set of students whose names aren't known in advance. I can define a class to store the names in a dictionary instead of using a predefined attribute for each student:

```
1  class SimpleGradebook:  
2      def __init__(self):  
3          self._grades = {}  
4      def add_student(self, name):  
5          self._grades[name] = []  
6      def report_grade(self, name, score):  
7          self._grades[name].append(score)  
8      def average_grade(self, name):  
9          grades = self._grades[name]  
10         return sum(grades) / len(grades)
```

```
1 book = SimpleGradebook()
2 book.add_student('Isaac Newton')
3 book.report_grade('Isaac Newton', 90)
4 book.report_grade('Isaac Newton', 95)
5 book.report_grade('Isaac Newton', 85)
6 print(book.average_grade('Isaac Newton'))
7 >>>
8 90.0
```

Dictionaries and their related built-in types are so easy to use that there's a danger of overextending them to write brittle code. For example, say that I want to extend the `SimpleGradebook` class to keep a list of grades by subject, not just overall. I can do this by changing the `_grades` dictionary to map student names (its keys) to yet another dictionary (its values). The innermost dictionary will map subjects (its keys) to a list of grades (its values). Here, I do this by using a `defaultdict` instance for the inner dictionary to handle missing subjects (see Item 17: "Prefer defaultdict Over setdefault to Handle Missing Items in Internal State" for background):

index

Creating object and classes # Python is an object-oriented language. In python everything is object i.e int, str, bool even modules, functions are al...

1. [Home](#)
2. [Blog](#)
3. Python Object and Classes

(Sponsors) Get started learning Python with DataCamp's free [Intro to Python tutorial](#). Learn Data Science by completing interactive coding challenges and watching videos by expert instructors. [Start Now!](#)

Updated on Jan 07, 2020

Creating object and classes

Python is an object-oriented language. In python everything is object i.e `int`, `str`, `bool` even modules, functions are also objects.

Object oriented programming use objects to create programs, and these objects stores data and behaviours.

Defining class

Class name in python is preceded with `class` keyword followed by a colon (`:`). Classes commonly contains data field to store the data and methods for defining behaviors. Also every class in python contains a special method called `initializer` (also commonly known as constructors), which get invoked automatically every time new object is created.

Let's see an example.

Here we have created a class called `Person` which contains one data field called `name` and method `whoami()`.

What is self?

All methods in python including some special methods like initializer have first parameter `self`. This parameter refers to the object which invokes the method. When you create new object the `self` parameter in the `__init__` method is automatically set to reference the object you have just created.

Creating object from class

Expected Output:

note:

When you call a method you don't need to pass anything to `self` parameter, python automatically does that for you behind the scenes.

You can also change the `name` data field.

Expected Output:

Although it is a bad practice to give access to your data fields outside the class. We will discuss how to prevent this next.

Hiding data fields #

To hide data fields you need to define private data fields. In python you can create private data field using two leading underscores. You can also define a private method using two leading underscores.

Let's see an example

Expected Output:

Let's try to access `__balance` data field outside of class.

Expected Output:

`AttributeError: 'BankAccount' object has no attribute '__balance'`

As you can see, now the `__balance` field is not accessible outside the class.

In next chapter we will learn about [operator overloading](#).

[Other Tutorials \(Sponsors\)](#)

This site generously supported by [DataCamp](#). DataCamp offers online interactive [Python Tutorials](#) for Data Science. Join over a million other learners and get started learning Python for data science today!

[Source](#)

Queue & Stacks

If you often work with lists in Python, then you probably know that they don't perform fast enough when you need to **pop** and **append** items on their left end. Python's `collections` module provides a class called `deque` that's specially designed to provide fast and memory-efficient ways to append and pop item from both ends of the underlying data structure.

Python's `deque` is a low-level and highly optimized **double-ended queue** that's useful for implementing elegant, efficient, and Pythonic queues and stacks, which are the most common list-like data types in computing.

In this tutorial, you'll learn:

- How to create and use Python's `deque` in your code
- How to efficiently **append** and **pop** items from both ends of a `deque`
- How to use `deque` to build efficient **queues** and **stacks**
- When it's worth using `deque` instead of `list`

To better understand these topics, you should know the basics of working with Python **lists**. It'll also be beneficial for you to have a general understanding of **queues** and **stacks**.

Finally, you'll write a few examples that walk you through some common use cases of `deque`, which is one of Python's most powerful data types.

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Getting Started With Python's `deque`

Appending items to and popping them from the right end of a Python list are normally efficient operations. If you use the **Big O notation** for **time complexity**, then you can say that they're $O(1)$. However, when Python needs to reallocate memory to grow the underlying list for accepting new items, these operations are slower and can become $O(n)$.

Additionally, appending and popping items on the left end of a Python list are known to be inefficient operations with $O(n)$ speed.

Since Python lists provide both operations with `.append()` and `.pop()`, they're usable as **stacks** and **queues**. However, the performance issues you saw before can significantly affect the overall performance of your applications.

Python's `deque` was the first data type added to the `collections` module back in [Python 2.4](#). This data type was specially designed to overcome the efficiency problems of `.append()` and `.pop()` in Python list.

Deques are sequence-like data types designed as a generalization of **stacks** and **queues**. They support memory-efficient and fast append and pop operations on both ends of the data structure.

Note: `deque` is pronounced as "deck." The name stands for **double-ended queue**.

Append and pop operations on both ends of a `deque` object are stable and equally efficient because deques are **implemented** as a **doubly linked list**. Additionally, append and pop operations on deques are also **thread safe** and memory efficient. These features make deques particularly useful for creating custom stacks and queues in Python.

Deques are also the way to go if you need to keep a list of last-seen items because you can restrict the maximum length of your deques. If you do so, then once a deque is full, it automatically discards items from one end when you append new items on the opposite end.

Here's a summary of the main characteristics of `deque`:

- Stores items of any **data type**
- Is a **mutable** data type

- Supports membership operations with the `in` operator
- Supports indexing, like in `a_deque[i]`
- Doesn't support slicing, like in `a_deque[0:2]`
- Supports built-in functions that operate on sequences and iterables, such as `len()`, `sorted()`, `reversed()`, and more
- Doesn't support in-place sorting
- Supports normal and reverse iteration
- Supports pickling with `pickle`
- Ensures fast, memory-efficient, and thread-safe pop and append operations on both ends

Creating `deque` instances is a straightforward process. You just need to import `deque` from `collections` and call it with an optional `iterable` as an argument:>>>

```

1 >>> from collections import deque
2
3 >>> # Create an empty deque
4 >>> deque()
5 deque([])
6
7 >>> # Use different iterables to create deques
8 >>> deque((1, 2, 3, 4))
9 deque([1, 2, 3, 4])
10
11 >>> deque([1, 2, 3, 4])
12 deque([1, 2, 3, 4])
13
14 >>> deque(range(1, 5))
15 deque([1, 2, 3, 4])
16
17 >>> deque("abcd")
18 deque(['a', 'b', 'c', 'd'])
19
20 >>> numbers = {"one": 1, "two": 2, "three": 3, "four": 4}
21 >>> deque(numbers.keys())
22 deque(['one', 'two', 'three', 'four'])
23
24 >>> deque(numbers.values())
25 deque([1, 2, 3, 4])
26
27 >>> deque(numbers.items())
28 deque([('one', 1), ('two', 2), ('three', 3), ('four', 4)])

```

If you instantiate `deque` without providing an `iterable` as an argument, then you get an empty deque. If you provide an input `iterable`, then `deque` initializes the new instance with data from it. The initialization goes from left to right using `deque.append()`.

The `deque` initializer takes the following two optional arguments:

1. `iterable` holds an iterable that provides the initialization data.
2. `maxlen` holds an integer `number` that specifies the maximum length of the deque.

As mentioned previously, if you don't supply an `iterable`, then you get an empty deque. If you supply a value to `maxlen`, then your deque will only store up to `maxlen` items.

Finally, you can also use unordered iterables, such as `sets`, to initialize your deques. In those cases, you won't have a predefined order for the items in the final deque. [Remove ads](#)

Popping and Appending Items Efficiently

The most important difference between `deque` and `list` is that the former allows you to perform efficient append and pop operations on both ends of the sequence. The `deque` class implements dedicated `.popleft()` and `.appendleft()` methods that operate on the left end of the sequence directly:>>>

```
1 >>> from collections import deque
2
3 >>> numbers = deque([1, 2, 3, 4])
4 >>> numbers.popleft()
5 1
6 >>> numbers.popleft()
7 2
8 >>> numbers
9 deque([3, 4])
10
11 >>> numbers.appendleft(2)
12 >>> numbers.appendleft(1)
13 >>> numbers
14 deque([1, 2, 3, 4])
```

Here, you use `.popleft()` and `.appendleft()` to remove and add values, respectively, to the left end of `numbers`. These methods are specific to the design of `deque`, and you won't find them in `list`.

Just like `list`, `deque` also provides `.append()` and `.pop()` methods to operate on the right end of the sequence. However, `.pop()` behaves differently:>>>

```
1 >>> from collections import deque
2
3 >>> numbers = deque([1, 2, 3, 4])
4 >>> numbers.pop()
5 4
6
7 >>> numbers.pop(0)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 TypeError: pop() takes no arguments (1 given)
```

Here, `.pop()` removes and returns the last value in the deque. The method doesn't take an index as an argument, so you can't use it to remove arbitrary items from your deques. You can only use it to remove and return the rightmost item.

As you learned earlier, `deque` is implemented as a **doubly linked list**. So, every item in a given deque holds a reference (**pointer**) to the next and previous item in the sequence.

Doubly linked lists make appending and popping items from either end light and efficient operations. That's possible because only the pointers need to be updated. As a result, both operations have similar performance, $O(1)$. They're also predictable performance-wise because there's no need for reallocating memory and moving existing items to accept new ones.

Appending and popping items from the left end of a regular Python list requires shifting all the items, which ends up being an $O(n)$ operation. Additionally, adding items to the right end of a list often requires Python to reallocate memory and copy the current items to the new memory location. After that, it can add the new items. This process takes longer to complete, and the append operation passes from being $O(1)$ to $O(n)$.

Consider the following performance tests for appending items to the left end of a sequence, `deque` vs `list`:

```
1 # time_append.py
2
3 from collections import deque
4 from time import perf_counter
5
6 TIMES = 10_000
7 a_list = []
8 a_deque = deque()
9
10 def average_time(func, times):
```

```

11     total = 0.0
12     for i in range(times):
13         start = perf_counter()
14         func(i)
15         total += (perf_counter() - start) * 1e9
16     return total / times
17
18 list_time = average_time(lambda i: a_list.insert(0, i), TIMES)
19 deque_time = average_time(lambda i: a_deque.appendleft(i), TIMES)
20 gain = list_time / deque_time
21
22 print(f"list.insert()      {list_time:.6} ns")
23 print(f"deque.appendleft() {deque_time:.6} ns  ({gain:.6}x faster)")

```

In this script, `average_time()` computes the average time that executing a function (`func`) a given number of `times` takes. If you [run the script](#) from your command line, then you get the following output:

```

1 $ python time_append.py
2 list.insert()      3735.08 ns
3 deque.appendleft() 238.889 ns  (15.6352x faster)

```

In this specific example, `.appendleft()` on a `deque` is several times faster than `.insert()` on a `list`. Note that `deque.appendleft()` is $O(1)$, which means that the execution time is constant. However, `list.insert()` on the left end of the list is $O(n)$, which means that the execution time depends on the number of items to process.

In this example, if you increment the value of `TIMES`, then you'll get higher time measurements for `list.insert()` but stable (constant) results for `deque.appendleft()`. If you'd like to try a similar performance test on pop operations for both deques and lists, then you can expand the exercise block below and compare your results to *Real Python's* after you're done.

Exercise: Test `deque.popleft()` vs `list.pop(0)` performanceShow/Hide

Solution: Test `deque.popleft()` vs `list.pop(0)` performanceShow/Hide

The `deque` data type was designed to guarantee efficient append and pop operations on either end of the sequence. It's ideal for approaching problems that require the implementation of queue and stack data structures in Python.

Accessing Random Items in a `deque`

Python's `deque` returns mutable sequences that work quite similarly to lists. Besides allowing you to append and pop items from their ends efficiently, deques provide a group of list-like methods and other sequence-like operations to work with items at arbitrary locations. Here are some of them:

Option	Description
<code>.insert(i, value)</code>	Insert an item <code>value</code> into a deque at index <code>i</code> .
<code>.remove(value)</code>	Remove the first occurrence of <code>value</code> , raising <code>ValueError</code> if the <code>value</code> doesn't exist.
<code>a_deque[i]</code>	Retrieve the item at index <code>i</code> from a deque.
<code>del a_deque[i]</code>	Remove the item at index <code>i</code> from a deque.

You can use these methods and techniques to work with items at any position inside a `deque` object. Here's how to do that:>>>

```

1 >>> from collections import deque

```

```
2
3 >>> letters = deque("abde")
4
5 >>> letters.insert(2, "c")
6 >>> letters
7 deque(['a', 'b', 'c', 'd', 'e'])
8
9 >>> letters.remove("d")
10 >>> letters
11 deque(['a', 'b', 'c', 'e'])
12
13 >>> letters[1]
14 'b'
15
16 >>> del letters[2]
17 >>> letters
18 deque(['a', 'b', 'e'])
```

Here, you first insert "c" into `letters` at position 2. Then you remove "d" from the deque using `.remove()`. Deques also allow **indexing** to access items, which you use here to access "b" at index 1. Finally, you can use the `del` keyword to delete any existing items from a deque. Note that `.remove()` lets you delete items *by value*, while `del` removes items *by index*.

Even though `deque` objects support indexing, they don't support **slicing**. In other words, you can't extract a **slice** from an existing deque using the **slicing syntax**, `[start:stop:step]`, as you would with a regular list:>>>

```
1 >>> from collections import deque
2
3 >>> numbers = deque([1, 2, 3, 4, 5])
4
5 >>> numbers[1:3]
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: sequence index must be integer, not 'slice'
```

Deques support indexing, but interestingly, they don't support slicing. When you try to get a slice from a deque, you get a `TypeError`. In general, performing a slicing on a linked list would be inefficient, so the operation isn't available.

So far, you've seen that `deque` is quite similar to `list`. However, while `list` is based on **arrays**, `deque` is based on a doubly linked list.

There is a hidden cost behind `deque` being implemented as a doubly linked list: accessing, inserting, and removing arbitrary items aren't efficient operations. To perform them, the **interpreter** has to iterate through the deque until it gets to the desired item. So, they're $O(n)$ instead of $O(1)$ operations.

Here's a script that shows how deques and lists behave when it comes to working with arbitrary items:

```
1 # time_random_access.py
2
3 from collections import deque
4 from time import perf_counter
5
6 TIMES = 10_000
7 a_list = [1] * TIMES
8 a_deque = deque(a_list)
9
10 def average_time(func, times):
11     total = 0.0
12     for _ in range(times):
13         start = perf_counter()
14         func()
15         total += (perf_counter() - start) * 1e6
16     return total / times
```

```

17
18 def time_it(sequence):
19     middle = len(sequence) // 2
20     sequence.insert(middle, "middle")
21     sequence[middle]
22     sequence.remove("middle")
23     del sequence[middle]
24
25 list_time = average_time(lambda: time_it(a_list), TIMES)
26 deque_time = average_time(lambda: time_it(a_deque), TIMES)
27 gain = deque_time / list_time
28
29 print(f"list {list_time:.6} µs ({gain:.6}x faster)")
30 print(f"deque {deque_time:.6} µs")

```

This script times inserting, deleting, and accessing items in the middle of a deque and a list. If you run the script, then you get an output that looks like the following:

```

1 $ python time_random_access.py
2 list 63.8658 µs (1.44517x faster)
3 deque 92.2968 µs

```

Deques aren't random-access data structures like lists. Therefore, accessing elements from the middle of a deque is less efficient than doing the same thing on a list. The main takeaway here is that deques aren't always more efficient than lists.

Python's `deque` is optimized for operations on either end of the sequence, so they're consistently better than lists in this regard. On the other hand, lists are better for random-access and fixed-length operations. Here are some of the differences between deques and lists in terms of performance:

Operation	deque	list
Accessing arbitrary items through indexing	$O(1)$	$O(n)$
Popping and appending items on the left end	$O(1)$	$O(n)$
Popping and appending items on the right end	$O(1)$	$O(1) + \text{reallocation}$
Inserting and deleting items in the middle	$O(n)$	$O(n)$

In the case of lists, `.append()` has amortized performance affected by memory reallocation when the interpreter needs to grow the list to accept new items. This operation requires copying all the current items to the new memory location, which significantly affects the performance.

This summary can help you choose the appropriate data type for the problem at hand. However, make sure to profile your code before switching from lists to deques. Both of them have their performance strengths. [Remove ads](#)

Building Efficient Queues With `deque`

As you already learned, `deque` is implemented as a double-ended queue that provides a generalization of **stacks** and **queues**. In this section, you'll learn how to use `deque` for implementing your own queue **abstract data types (ADT)** at a low level in an elegant, efficient, and Pythonic way.

Note: In the Python standard library, you'll find `queue`. This module implements multi-producer, multi-consumer queues that allow you to exchange information between multiple threads safely.

If you're working with queues, then favor using those high-level abstractions over `deque` unless you're implementing your own data structure.

Queues are [collections](#) of items. You can modify queues by adding items at one end and removing items from the opposite end.

Queues manage their items in a **First-In/First-Out (FIFO)** fashion. They work as a pipe where you push in new items at one end of the pipe and pop old items out from the other end. Adding an item to one end of a queue is known as an **enqueue** operation. Removing an item from the other end is called **dequeue**.

To better understand queues, take your favorite restaurant as an example. The restaurant has a queue of people waiting for a table to order their food. Typically, the last person to arrive will stand at the end of the queue. The person at the beginning of the queue will leave it as soon as a table is available.

Here's how you can emulate the process using a bare-bones `deque` object:>>>

```
1 >>> from collections import deque
2
3 >>> customers = deque()
4
5 >>> # People arriving
6 >>> customers.append("Jane")
7 >>> customers.append("John")
8 >>> customers.append("Linda")
9
10 >>> customers
11 deque(['Jane', 'John', 'Linda'])
12
13 >>> # People getting tables
14 >>> customers.popleft()
15 'Jane'
16 >>> customers.popleft()
17 'John'
18 >>> customers.popleft()
19 'Linda'
20
21 >>> # No people in the queue
22 >>> customers.popleft()
23 Traceback (most recent call last):
24   File "<stdin>", line 1, in <module>
25 IndexError: pop from an empty deque
```

Here, you first create an empty `deque` object to represent the queue of people arriving at the restaurant. To enqueue a person, you use `.append()`, which adds individual items to the right end. To dequeue a person, you use `.popleft()`, which removes and returns individual items on the left end of a deque.

Cool! Your queue simulation works! However, since `deque` is a generalization, its API doesn't match the typical queue API. For example, instead of `.enqueue()`, you have `.append()`. You also have `.popleft()` instead of `.dequeue()`. Additionally, `deque` provides several other operations that might not fit your specific needs.

The good news is that you can create custom queue classes with the functionality you need and nothing else. To do this, you can internally use a deque to store the data and provide the desired functionality in your custom queues. You can think of it as an implementation of the [adapter design pattern](#), in which you convert the deque's interface into something that looks more like a queue interface.

For example, say you need a custom queue abstract data type that provides only the following features:

- Enqueuing items
- Dequeueing items
- Returning the length of the queue
- Supporting membership tests

- Supporting normal and reverse iteration
- Providing a user-friendly string representation

In this case, you can write a `Queue` class that looks like the following:

```

1 # custom_queue.py
2
3 from collections import deque
4
5 class Queue:
6     def __init__(self):
7         self._items = deque()
8
9     def enqueue(self, item):
10        self._items.append(item)
11
12    def dequeue(self):
13        try:
14            return self._items.popleft()
15        except IndexError:
16            raise IndexError("dequeue from an empty queue") from None
17
18    def __len__(self):
19        return len(self._items)
20
21    def __contains__(self, item):
22        return item in self._items
23
24    def __iter__(self):
25        yield from self._items
26
27    def __reversed__(self):
28        yield from reversed(self._items)
29
30    def __repr__(self):
31        return f"Queue({list(self._items)})"

```

Here, `_items` holds a `deque` object that allows you to store and manipulate the items in the queue. `Queue` implements `.enqueue()` using `deque.append()` to add items to the end of the queue. It also implements `.dequeue()` with `deque.popleft()` to efficiently remove items from the beginning of the queue.

The **special methods** support the following features:

Method	Support
<code>__len__()</code>	Length with <code>len()</code>
<code>__contains__()</code>	Membership tests with <code>in</code>
<code>__iter__()</code>	Normal iteration
<code>__reversed__()</code>	Reverse iteration
<code>__repr__()</code>	String representation

Ideally, `__repr__()` should return a string representing a valid Python expression. This expression will allow you to recreate the object unambiguously with the same value.

However, in the example above, the intent is to use the method's `return` value to gracefully display the object on the **interactive shell**. You can make it possible to build `queue` instances from this specific string representation by accepting an initialization iterable as an argument to `__init__()` and building instances from it.

With these final additions, your `Queue` class is complete. To use this class in your code, you can do something like the following:>>>

```
1 >>> from custom_queue import Queue
2
3 >>> numbers = Queue()
4 >>> numbers
5 Queue([])
6
7 >>> # Enqueue items
8 >>> for number in range(1, 5):
9 ...     numbers.enqueue(number)
10 ...
11 >>> numbers
12 Queue([1, 2, 3, 4])
13
14 >>> # Support len()
15 >>> len(numbers)
16 4
17
18 >>> # Support membership tests
19 >>> 2 in numbers
20 True
21 >>> 10 in numbers
22 False
23
24 >>> # Normal iteration
25 >>> for number in numbers:
26 ...     print(f"Number: {number}")
27 ...
28 1
29 2
30 3
31 4
```

As an exercise, you can test the remaining features and implement other features, such as supporting equality tests, removing and accessing random items, and more. Go ahead and give it a try! [Remove ads](#)

Exploring Other Features of `deque`

In addition to the features you've seen so far, `deque` also provides other methods and attributes specific to their internal design. They add new and useful functionalities to this versatile data type.

In this section, you'll learn about other methods and attributes that deques provide, how they work, and how to use them in your code.

Limiting the Maximum Number of Items: `maxlen`

One of the most useful features of `deque` is the possibility to specify the **maximum length** of a given deque using the `maxlen` argument when you're instantiating the class.

If you supply a value to `maxlen`, then your deque will only store up to `maxlen` items. In this case, you have a **bounded deque**. Once a bounded deque is full with the specified number of items, adding a new item at either end automatically removes and discards the item at the opposite end:>>>

```
1 >>> from collections import deque
2
3 >>> four_numbers = deque([0, 1, 2, 3, 4], maxlen=4) # Discard 0
4 >>> four_numbers
5 deque([1, 2, 3, 4], maxlen=4)
6
7 >>> four_numbers.append(5) # Automatically remove 1
8 >>> four_numbers
9 deque([2, 3, 4, 5], maxlen=4)
```

```

10
11 >>> four_numbers.append(6) # Automatically remove 2
12 >>> four_numbers
13 deque([3, 4, 5, 6], maxlen=4)
14
15 >>> four_numbers.appendleft(2) # Automatically remove 6
16 >>> four_numbers
17 deque([2, 3, 4, 5], maxlen=4)
18
19 >>> four_numbers.appendleft(1) # Automatically remove 5
20 >>> four_numbers
21 deque([1, 2, 3, 4], maxlen=4)
22
23 >>> four_numbers maxlen
24 4

```

If the number of items in the input iterable is greater than `maxlen`, then `deque` discards the left-most items (`0` in the example). Once the deque is full, appending an item on any end automatically removes the item on the other end.

Note that if you don't specify a value to `maxlen`, then it defaults to `None`, and the deque can grow to an arbitrary number of items.

Having the option to restrict the maximum number of items allows you to use deques for tracking the latest elements in a given sequence of objects or events. For example, you can track the last five transactions in a bank account, the last ten open text files in an editor, the last five pages in a browser, and more.

Note that `maxlen` is available as a read-only attribute in your deques, which allows you to check if the deque is full, like in `deque maxlen == len(deque)`.

Finally, you can set `maxlen` to any positive integer number representing the maximum number of items you want to store in a specific deque. If you supply a negative value to `maxlen`, then you get a `ValueError`.

Rotating the Items: `.rotate()`

Another interesting feature of deques is the possibility to rotate their elements by calling `.rotate()` on a non-empty deque. This method takes an integer `n` as an argument and rotates the items `n` steps to the right. In other words, it moves `n` items from the right end to the left end in a circular fashion.

The default value of `n` is `1`. If you provide a negative value to `n`, then the rotation is to the left:>>>

```

1 >>> from collections import deque
2
3 >>> ordinals = deque(["first", "second", "third"])
4
5 >>> # Rotate items to the right
6 >>> ordinals.rotate()
7 >>> ordinals
8 deque(['third', 'first', 'second'])
9
10 >>> ordinals.rotate(2)
11 >>> ordinals
12 deque(['first', 'second', 'third'])
13
14 >>> # Rotate items to the left
15 >>> ordinals.rotate(-2)
16 >>> ordinals
17 deque(['third', 'first', 'second'])
18
19 >>> ordinals.rotate(-1)
20 >>> ordinals
21 deque(['first', 'second', 'third'])

```

In these examples, you rotate `ordinals` several times using `.rotate()` with different values of `n`. If you call `.rotate()` without an argument, then it relies on the default value of `n` and rotates the deque `1` position to the right. Calling the method with a negative `n` allows you to rotate the items to the left.

Adding Several Items at Once: `.extendleft()`

Like regular lists, deques provide an `.extend()` method, which allows you to add several items to the right end of a deque using an `iterable` as an argument. Additionally, deques have a method called `extendleft()`, which takes an `iterable` as an argument and adds its items to the left end of the target deque in one go:>>>

```
1 >>> from collections import deque
2
3 >>> numbers = deque([1, 2])
4
5 >>> # Extend to the right
6 >>> numbers.extend([3, 4, 5])
7 >>> numbers
8 deque([1, 2, 3, 4, 5])
9
10 >>> # Extend to the left
11 >>> numbers.extendleft([-1, -2, -3, -4, -5])
12 >>> numbers
13 deque([-5, -4, -3, -2, -1, 1, 2, 3, 4, 5])
```

Calling `.extendleft()` with an `iterable` extends the target deque to the left. Internally, `.extendleft()` performs a series of individual `.appendleft()` operations that process the input iterable from left to right. This ends up adding the items in reverse order to the left end of the target deque. Remove ads

Using Sequence-Like Features of `deque`

Since deques are mutable sequences, they implement almost all the methods and operations that are common to `sequences` and `mutable sequences`. So far, you've learned about some of these methods and operations, such as `.insert()`, indexing, membership tests, and more.

Here are a few examples of other actions you can perform on `deque` objects:>>>

```
1 >>> from collections import deque
2
3 >>> numbers = deque([1, 2, 2, 3, 4, 4, 5])
4
5 >>> # Concatenation
6 >>> numbers + deque([6, 7, 8])
7 deque([1, 2, 2, 3, 4, 4, 5, 6, 7, 8])
8
9 >>> # Repetition
10 >>> numbers * 2
11 deque([1, 2, 2, 3, 4, 4, 5, 1, 2, 2, 3, 4, 4, 5])
12
13 >>> # Common sequence methods
14 >>> numbers = deque([1, 2, 2, 3, 4, 4, 5])
15 >>> numbers.index(2)
16 1
17 >>> numbers.count(4)
18 2
19
20 >>> # Common mutable sequence methods
21 >>> numbers.reverse()
22 >>> numbers
23 deque([5, 4, 4, 3, 2, 2, 1])
24
25 >>> numbers.clear()
```

```
26 >>> numbers
27 deque([])
```

You can use the addition operator (`+`) to concatenate two existing deques. On the other hand, the multiplication operator (`*`) returns a new deque equivalent to repeating the original deque as many times as you want.

Regarding other sequence methods, the following table provides a summary:

Method	Description
<code>.clear()</code>	Remove all the elements from a deque.
<code>.copy()</code>	Create a shallow copy of a deque.
<code>.count(value)</code>	Count the number times <code>value</code> appears in a deque.
<code>.index(value)</code>	Return the position of <code>value</code> in the deque.
<code>.reverse()</code>	Reverse the elements of the deque in place and then return <code>None</code> .

Here, `.index()` can also take two optional arguments: `start` and `stop`. They allow you to restrict the search to those items at or after `start` and before `stop`. The method raises a `ValueError` if `value` doesn't appear in the deque at hand.

Unlike lists, deques don't include a `.sort()` method to sort the sequence in place. This is because sorting a linked list would be an inefficient operation. If you ever need to sort a deque, then you can still use `sorted()`.

Putting Python's `deque` Into Action

You can use deques in a fair amount of use cases, such as to implement queues, stacks, and [circular buffers](#). You can also use them to maintain an undo-redo history, enqueue incoming requests to a [web service](#), keep a list of recently open files and websites, safely exchange data between multiple threads, and more.

In the following sections, you'll code a few small examples that will help you better understand how to use deques in your code.

Keeping a Page History

Having a `maxlen` to restrict the maximum number of items makes `deque` suitable for solving several problems. For example, say you're building an application that [scrapes](#) data from search engines and social media sites. At some point, you need to keep track of the three last sites your application requested data from.

To solve this problem, you can use a deque with a `maxlen` of `3` :>>>

```
1 >>> from collections import deque
2
3 >>> sites = (
4 ...     "google.com",
5 ...     "yahoo.com",
6 ...     "bing.com"
7 ... )
8
9 >>> pages = deque(maxlen=3)
10 >>> pages maxlen
11 3
12
13 >>> for site in sites:
14 ...     pages.appendleft(site)
15 ...
16
17 >>> pages
```

```
18 deque(['bing.com', 'yahoo.com', 'google.com'], maxlen=3)
19
20 >>> pages.appendleft("facebook.com")
21 >>> pages
22 deque(['facebook.com', 'bing.com', 'yahoo.com'], maxlen=3)
23
24 >>> pages.appendleft("twitter.com")
25 >>> pages
26 deque(['twitter.com', 'facebook.com', 'bing.com'], maxlen=3)
```

In this example, `pages` keeps a list of the last three sites your application visited. Once `pages` is full, adding a new site to an end of the deque automatically discards the site at the opposite end. This behavior keeps your list up to date with the last three sites you used.

Note that you can set `maxlen` to any positive integer representing the number of items to store in the deque at hand. For example, if you want to keep a list of ten sites, then you can set `maxlen` to `10`.

Sharing Data Between Threads

Python's `deque` is also useful when you're coding [multithreaded](#) applications, as described by [Raymond Hettinger](#), core Python developer and creator of `deque` and the `collections` module:

The deque's `.append()`, `.appendleft()`, `.pop()`, `.popleft()`, and `len(d)` operations are thread-safe in CPython. ([Source](#))

Because of that, you can safely add and remove data from both ends of a deque at the same time from separate threads without the risk of data corruption or other associated issues.

To try out how `deque` works in a multithreaded application, fire up your favorite [code editor](#), create a new script called `threads.py`, and add the following code to it:

```
1 # threads.py
2
3 import logging
4 import random
5 import threading
6 import time
7 from collections import deque
8
9 logging.basicConfig(level=logging.INFO, format"%(message)s")
10
11 def wait_seconds(mins, maxs):
12     time.sleep(mins + random.random() * (maxs - mins))
13
14 def produce(queue, size):
15     while True:
16         if len(queue) < size:
17             value = random.randint(0, 9)
18             queue.append(value)
19             logging.info("Produced: %d -> %s", value, str(queue))
20         else:
21             logging.info("Queue is saturated")
22             wait_seconds(0.1, 0.5)
23
24 def consume(queue):
25     while True:
26         try:
27             value = queue.popleft()
28         except IndexError:
29             logging.info("Queue is empty")
30         else:
31             logging.info("Consumed: %d -> %s", value, str(queue))
32             wait_seconds(0.2, 0.7)
33
34 logging.info("Starting Threads...\n")
35 logging.info("Press Ctrl+C to interrupt the execution\n")
36
37 shared_queue = deque()
```

```
38
39     threading.Thread(target=produce, args=(shared_queue, 10)).start()
40     threading.Thread(target=consume, args=(shared_queue,)).start()
```

Here, `produce()` takes a `queue` and a `size` as arguments. Then it uses `random.randint()` in a `while` loop to continuously produce `random` numbers and store them in a `global` deque called `shared_queue`. Since appending items to a deque is a thread-safe operation, you don't need to use a `lock` to protect the shared data from other threads.

The helper function `wait_seconds()` simulates that both `produce()` and `consume()` represent long-running operations. It returns a random wait-time value between a given range of seconds, `mins` and `maxs`.

In `consume()`, you call `.popLeft()` inside a loop to systematically retrieve and remove data from `shared_queue`. You wrap the call to `.popLeft()` in a `try ... except` statement to handle those cases in which the shared queue is empty.

Note that while you defined `shared_queue` in the global `namespace`, you access it through local variables inside `produce()` and `consume()`. Accessing the global variable directly would be more problematic and definitely not a best practice.

The final two lines in the script create and start separate threads to execute `produce()` and `consume()` concurrently. If you run the script from your command line, then you'll get an output similar to the following:

```
1 $ python threads.py
2 Starting Threads...
3
4 Press Ctrl+C to interrupt the execution
5
6 Produced: 1 -> deque([1])
7 Consumed: 1 -> deque([])
8 Queue is empty
9 Produced: 3 -> deque([3])
10 Produced: 0 -> deque([3, 0])
11 Consumed: 3 -> deque([0])
12 Consumed: 0 -> deque([])
13 Produced: 1 -> deque([1])
14 Produced: 0 -> deque([1, 0])
15 ...
```

The producer thread adds numbers to the right end of the shared deque, while the consumer thread consumes numbers from the left end. To interrupt the script execution, you can press `Ctrl+C` on your keyboard.

Finally, you can play a little bit with the time interval inside `produce()` and `consume()`. Change the values you pass to `wait_seconds()`, and watch how the program behaves when the producer is slower than the consumer and the other way around.

[Remove ads](#)

Emulating the `tail` Command

The final example you'll code here emulates the `tail` command, which is available on `Unix` and `Unix-like` operating systems. The command accepts a file path at the command line and prints the last ten lines of that file to the system's standard output. You can tweak the number of lines you need `tail` to print with the `-n`, `--lines` option.

Here's a small Python function that emulates the core functionality of `tail` >>>

```
1 >>> from collections import deque
2
3 >>> def tail(filename, lines=10):
4     ...     try:
5     ...         with open(filename) as file:
6     ...             return deque(file, lines)
```

```
7 ...     except OSError as error:  
8 ...         print(f'Opening file "{filename}" failed with error: {error}')  
9 ...
```

Here, you define `tail()`. The first argument, `filename`, holds the path to the target file as a `string`. The second argument, `lines`, represents the number of lines you want to retrieve from the end of the target file. Note that `lines` defaults to `10` to simulate the default behavior of `tail`.

Note: The original idea for this example comes from the Python documentation on `deque`. Check out the section on `deque recipes` for further examples.

The `deque` in the highlighted line can only store up to the number of items you pass to `lines`. This guarantees that you get the desired number of lines from the end of the input file.

As you saw before, when you create a bounded `deque` and initialize it with an iterable the contains more items than allowed (`maxlen`), the `deque` constructor discards all the leftmost items in the input. Because of that, you end up with the last `maxlen` lines of the target file.

Conclusion

Queues and stacks are commonly used **abstract data types** in programming. They typically require efficient `pop` and `append` operations on either end of the underlying data structure. Python's `collections` module provides a data type called `deque` that's specially designed for fast and memory-efficient append and pop operations on both ends.

With `deque`, you can code your own queues and stacks at a low level in an elegant, efficient, and Pythonic way.

In this tutorial, you learned how to:

- Create and use Python's `deque` in your code
- Efficiently `append` and `pop` items from both ends of a sequence with `deque`
- Use `deque` to build efficient **queues** and **stacks** in Python
- Decide when to use `deque` instead of `list`

In this tutorial, you also coded a few examples that helped you approach some common use cases of `deque` in Python.

MISC

Outline

Week 17

Data Structures

Keywords:

```
***and      del      for      is      raiseassert      elif      from      lambda      returnbreak      else      gl
```

[py-notes.pdf](#)

<https://bryan-guner.gitbook.io/notesarchive/>

<https://docs.python.org/3/>

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):                  # error: first line indented
2  for i in range(len(l)):        # error: not indented
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	Comparison operators	Dictionary operations
<pre>+ - subtraction * / division ** exponent % // floor division</pre>	<pre>== equal != different > higher < lower >= higher or equal <= lower or equal</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>
Boolean operators	Special characters	String methods
<pre>and logical AND or logical OR not logical NOT</pre>	<pre># coment \n new line \<char> escape char</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>
String operations		Dictionary methods
<pre>string[i] retrieves character at position i string[-1] retrieves last character string[i:j] retrieves characters in range i to j</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>

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 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 to the end of a string.
4 print("Spaghetti"[1:]) # => spaghetti
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 that
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 Billy Bob
```

- 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>&&</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:

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 types.
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 to the issue
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 to the issue
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))
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 for a user input.
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 and purpose. Programs typically have
```

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 separated 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 if all items in the iterable are true.

Working with dictionaries

`dir(dictionary)` : returns the list of keys in the dictionary.

- *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
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 elements that appear in the first set and
```

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
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('Carlito 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 and 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/82154f50603f73826c27377ebaa498b5#file-python-study-guide-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

2021-03-06_Python-Study-Guide-for-a-JavaScript-Programmer-

Built-in Types

Super Simple Intro To Python

D1



gist.md

<https://gist.github.com/bgoonz/fdc61c788821939f20d2ec231331cde4>



gist.md

<https://gist.github.com/bgoonz/fdc61c788821939f20d2ec231331cde4>

Week 17

Data Structures

Keywords:

```
***and      del      for      is      raiseassert      elif      from      lambda      returnbreak      else      gl
```

[py-notes.pdf](#)

<https://bryan-guner.gitbook.io/notesarchive/>

<https://docs.python.org/3/>

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):          # error: first line indented
2  for i in range(len(l)): # error: not indented
```

```

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

Main data types		List operations		List methods	
<code>boolean = True / False</code>		<code>list = []</code>	defines an empty list	<code>list.append(x)</code>	adds x to the end of the list
<code>integer = 10</code>		<code>list[i] = x</code>	stores x with index i	<code>list.extend(L)</code>	appends L to the end of the list
<code>float = 10.01</code>		<code>list[i]</code>	retrieves the item with index i	<code>list.insert(i,x)</code>	inserts x at i position
<code>string = "123abc"</code>		<code>list[-1]</code>	retrieves last item	<code>list.remove(x)</code>	removes the first list item whose value is x
<code>list = [value1, value2, ...]</code>		<code>list[i:j]</code>	retrieves items in the range i to j	<code>list.pop(i)</code>	removes the item at position i and returns its value
<code>dictionary = { key1:value1, key2:value2, ...}</code>		<code>del list[i]</code>	removes the item with index i	<code>list.clear()</code>	removes all items from the list
Numeric operators		Dictionary operations		String methods	
<code>+</code> addition	<code>==</code> equal	<code>dict = {}</code>	defines an empty dictionary	<code>string.upper()</code>	converts to uppercase
<code>-</code> subtraction	<code>!=</code> different	<code>dict[k] = x</code>	stores x associated to key k	<code>string.lower()</code>	converts to lowercase
<code>*</code> multiplication	<code>></code> higher	<code>dict[k]</code>	retrieves the item with key k	<code>string.count(x)</code>	counts how many times x appears
<code>/</code> division	<code><</code> lower	<code>del dict[k]</code>	removes the item with key k	<code>string.find(x)</code>	position of the x first occurrence
<code>**</code> exponent	<code>>=</code> higher or equal			<code>string.replace(x,y)</code>	replaces x for y
<code>%</code> modulus				<code>string.strip(x)</code>	returns a list of values delimited by x
<code>//</code> floor division	<code><=</code> lower or equal			<code>string.join(L)</code>	returns a string with L values joined by string
Boolean operators		Special characters		<code>string.format(x)</code>	returns a string that includes formatted x
<code>and</code> logical AND	<code>#</code> coment				
<code>or</code> logical OR	<code>\n</code> new line				
<code>not</code> logical NOT	<code>\<char></code> escape char				
String operations		Dictionary methods			
<code>string[i]</code>	retrieves character at position i			<code>dict.keys()</code>	returns a list of keys
<code>string[-1]</code>	retrieves last character			<code>dict.values()</code>	returns a list of values
<code>string[i:j]</code>	retrieves characters in range i to j			<code>dict.items()</code>	returns a list of pairs (key,value)
				<code>dict.get(k)</code>	returns the value associated to the key k
				<code>dict.pop()</code>	removes the item associated to the key and returns its value
				<code>dict.update(D)</code>	adds keys-values (D) to dictionary
				<code>dict.clear()</code>	removes all keys-values from the dictionary
				<code>dict.copy()</code>	returns a copy of the dictionary

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 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 to the end of a string.
4 print("Spaghetti")[1:] # => spaghetti
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 that
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.**
- 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 Billy Bob
```

- 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>&&</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:

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 types.
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 to the issue
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 to the issue
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 # 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 for a user input.
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 and purpose. Programs typically have
```

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.

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 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}
4 Difference : The - operator can be used to produce a new set of only the elements that appear in the first set and
```

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
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/82154f50603f73826c27377ebaa498b5#file-python-study-guide-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

<https://gist.github.com/bgoonz/282774d28326ff83d8b42ae77ab1fee3#file-python-cheatsheet-py>

[2021-03-06_Python-Study-Guide-for-a-JavaScript-Programmer-](#)

[Built-in Types](#)

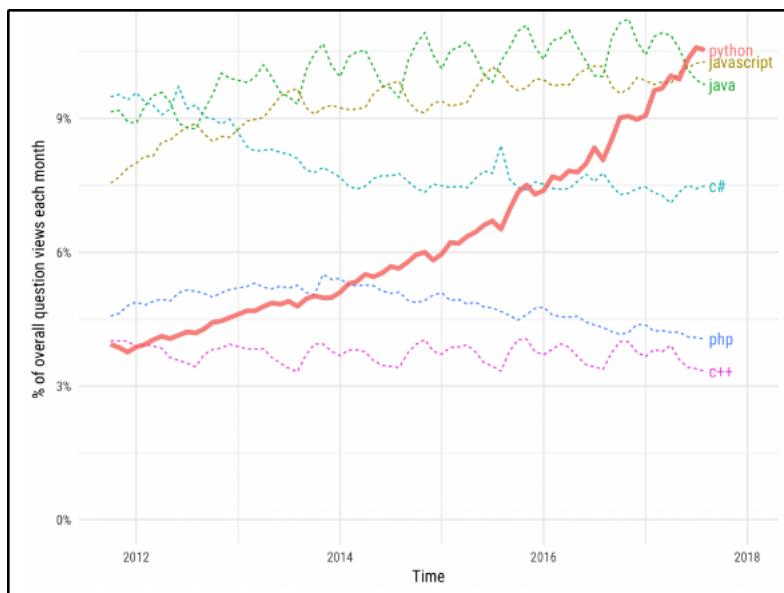
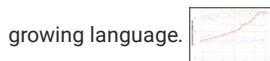
[Super Simple Intro To Python](#)

[D1](#)

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!')
```

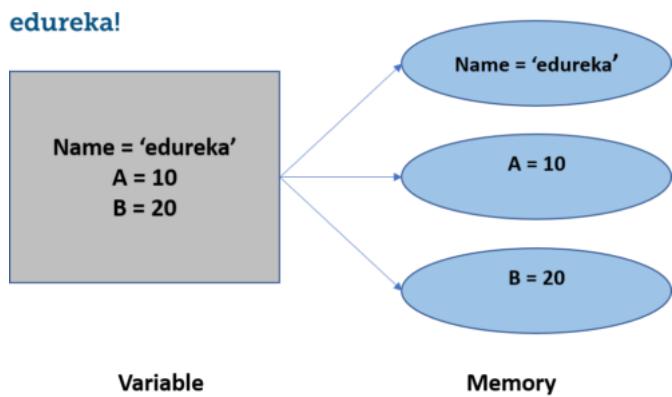
Output: Hello World, Welcome to edureka!

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)

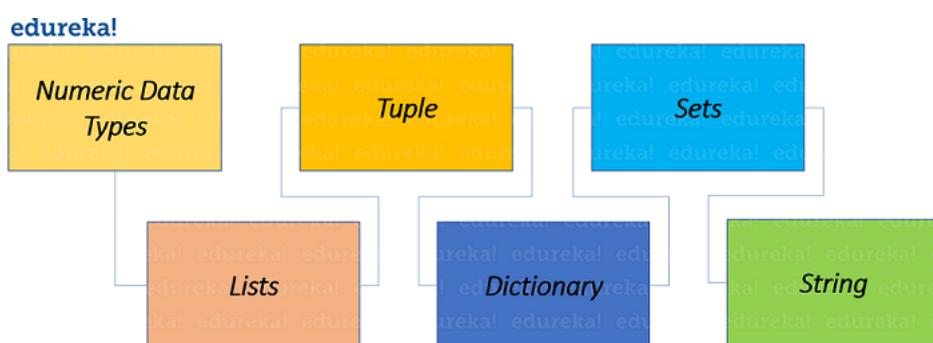
```

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 ($2.5\text{e}2 = 2.5 \times 10^2 = 250$). 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
a[2]	135	Finds the data at index 2 and returns it.
a[0:3]	[3.142, 'Hindi', 135]	Data from index 0 to 2 is returned as the last index mentioned is always ignored.
a[3] = 'edureka!'	moves 'edureka!' to index 3	The data is replaced in index 3
del a[1]	Deletes 'Hindi' from the list	Delete items and it does not return any item back
len(a)	3	Obtain the length of a variable in Python
a * 2	Output the list 'a' twice	If a dictionary is multiplied with a number, it is repeated that many number of times
a[::-1]	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.
a.append(3)	3 will be added at the end of the list	Add data at the end of the list
a.extend(b)	[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'.
a.insert(3,'hello')	[3.142, 135, 'edureka!', 'hello', 3, 2]	Takes the index and the value and adds value to that index.
a.remove(3.142)	[135, 'edureka!', 'hello', 3, 2]	Removes the value from the list that has been passed as an argument. No value returned.
a.index(135)	0	Finds the element 135 and returns the index of that data
a.count('hello')	1	It goes through the string and finds the times it has been repeated in the list
a.pop(1)	'edureka!'	Pops the element in the given index and returns the element if needed.
a.reverse()	[2, 3, 'hello', 135]	It just reverses the list
a.sort()	[5, 1234, 64738]	Sorts the list based on ascending or descending order.
a.clear()	[]	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', 'ankita@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 **unordered 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.

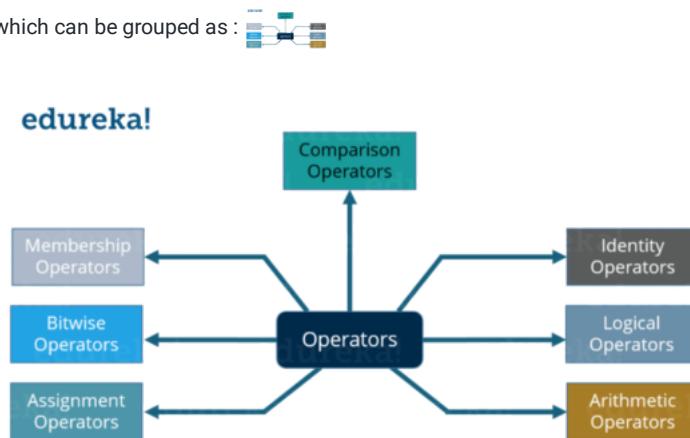
That basically sums up the data types in Python. I hope you have a good understanding of the same and if you have any doubts, please leave a comment and I will get back to you as soon as possible.

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 :



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 NOT 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

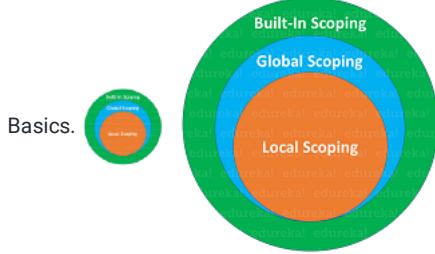
Namespacing works with scopes. **Scopes** are the *validity of a function/variable/value inside the function or class they belong to*. Python **built-in functions** namespacing **covers all the other scopes of Python**. Functions such as print() and id() etc. can be used even without any imports and 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.

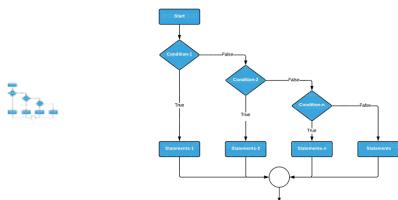
You can get a better understanding of the scopes and namespacing from the figure below. The **built-in scope** covers all of Python making them *available whenever needed*. The **global scope** covers all of the *programs* that are being executed. The **local scope** covers all of the

methods being executed in a program. That is basically what namespaces are in Python. Let us move ahead with flow control in Python



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
```

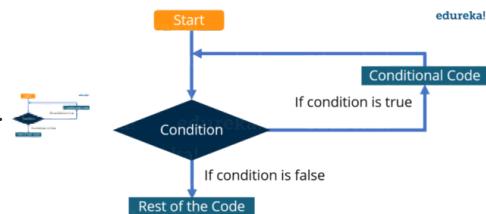
This means that if a condition is met, do something. Else go through the remaining elif conditions and finally if no condition is met, execute the else block. You can even have nested if-else statements inside the if-else blocks.

```
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
999999999
```

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 current 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.

Python VS JavaScript

Contents

- Versions
- Development Environments
- Running Programs
- Comments
- Semicolons
- Whitespace, Blocks
- Functions
- Arithmetic Operators
- Variables
- Data Types
- Arrays/Lists
- Slices
- Objects/Dicts
- String Formatting
- Booleans and Conditionals
- `for` Loops
- `while` Loops
- `switch` Statement
- `if` Conditionals
- Classes

Versions

JavaScript

The standard defining JavaScript (JS) is *ECMAScript (ES)*. Modern browsers and NodeJS support ES6, which has a rich feature set. Older browsers might not support all ES6 features.

The website caniuse.com will show which browsers support specific JS features.

Python

Python 3.x is the current version, but there are a number of packages and sites running legacy Python 2.

On some systems, you might have to be explicit when you invoke Python about which version you want by running `python2` or `python3`. The `--version` command line switch will tell you which version is running. Example:

```
1 $ python --version
2 Python 2.7.10
3 $ python2 --version
4 -bash: python2: command not found
5 $ python3 --version
6 Python 3.6.5
```

Using `virtualenv` or `pipenv` can really ease development painpoints surrounding the version. See Development Environments, below.

Development Environments

JavaScript

For managing project packages, the classic tool is `npm`. This is slowly being superseded by the newer `yarn` tool. Choose one for a project, and don't mix and match.

Python

For managing project packages and Python versions, the classic tool is `virtualenv`. This is slowly being superseded by the newer `pipenv` tool.

Running Programs

JavaScript

Running from the command line with NodeJS:

```
node program.js arg1 arg2 etc
```

In a web page, a script is referenced with a `<script>` HTML tag:

```
<script src="program.js"></script>
```

Python

Running from the command line:

```
python program.py arg1 arg2 etc
```

Comments

JavaScript

Single line:

```
1 // Anything after two forward slashes is a comment
2 print(2); // prints 2
```

Multi-line comments:

```
1 /* Anything between slash-star and
2 star-slash is a comment */
```

You may not nest multi-line comments.

Python

Single line:

```
1 # Anything after a # is a comment
2 print(2) # prints 2
```

Python doesn't directly support multi-line comments, but you can effectively do them by using multi-line strings `"""` :

```
1 """
2 At this point we wish
3 to print out some numbers
4 and see where that gets us
5 """
6 print(1)
7 print(2)
```

Semicolons

JavaScript

Javascript ends statements with semicolons, usually at the end of the line. It can also be effectively used to put multiple statements on the same line, but this is rare.

```
1 console.log("Hello, world!");
2
3 let x = 10; console.log(x);
```

Javascript interpreters will let you get away without using semicolons at ends of lines, but you should use them.

Python

Python can separate statements by semicolons, though this is rare in practice.

```
print(1); print(2) # prints 1, then 2
```

Whitespace, blocks

JavaScript

Whitespace has no special meaning. Blocks are declared with curly braces `{` and `}`.

```
1 if (x == 2) {
2   console.log("x must be 2")
3 } else {
4   if (x == 3) {
5     console.log("x must be 3")
6   } else {
7     console.log("x must be something else")
8   }
9 }
```

Python

Indentation level is how blocks are declared. The preferred method is to use spaces, but tabs can also be used.

```
1 if x == 2:
2     print("x must be 2")
3 else:
4     if x == 3:
5         print("x must be 3")
6     else:
7         print("x must be something else")
```

Functions

JavaScript

Define functions as follows:

```
1 function foobar(x, y, z) {
2     console.log(x, y, z);
3     return 12;
4 }
```

An alternate syntax for functions is growing increasingly common, called *arrow functions*:

```
1 let hello = () => {
2     console.log("hello");
3     console.log("world");
4 }
5
6 hello(); // prints hello, then world
7
8 // Arrow functions with single parameters don't
9 // need parens around the parameter:
10 let printNum = x => {
11     console.log(x);
12 }
13
14 // If you don't explicitly return a value, the value
15 // of the last expression in the function is used.
16 // Also, single-expression functions don't need
17 // braces around them.
18 let add = (x, y) => x + y;
19
20 console.log(add(4, 5)); // prints 9
```

Python

Define functions as follows:

```
1 def foobar(x, y, z):
2     print(x, y, z)
3     return 12
```

Python also supports the concept of *lambda functions*, which are simple functions that can do basic operations.

```
1 add = lambda x, y: x + y
2
3 print(add(4, 5)) # prints 9
```

Arithmetic Operators

JavaScript

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
--	Pre-decrement, post-decrement
++	Pre-increment, post-increment
**	Exponentiation (power)
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulo assignment

Python

The pre- and post-increment and decrement are notably absent.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
**	Exponentiation (power)
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment

%=

Modulo assignment

Variables

Javascript

Variables are created upon use, but should be created with the `let` or `const` keywords.

```
1 let x = 10;
2 const y = 30;
```

`var` is an outdated way of declaring variables in Javascript.

Python

Variables are created upon use.

```
x = 10
```

Data Types

JavaScript

```
1 let a = 12;           // number
2 let b = 1.2;          // number
3 let c = 'hello';      // string
4 let d = "world";      // string
5 let e = true;          // boolean
6 let f = null;          // null value
7 let g = undefined;    // undefined value
```

Multi-line strings:

```
1 let s = `this is a
2 multi-line string`;
```

Parameterized strings:

```
1 let x = 12;
2 console.log(`x is ${x}`); // prints "x is 12"
```

JS is *weakly typed* so it supports operations on multiple types of data at once.

```
1 "2" + 4;           // string "24"
2 parseInt("2") + 4; // number 6
3 Number("2") + 4;  // number 6
```

Python

```
1 a = 12      # int (integer)
2 b = 1.2     # float (floating point)
3 c = 'hello' # str (string)
4 d = "world" # str
5 e = False   # bool (boolean)
6 f = None    # null value
```

Multi-line strings:

```
1 s = """this is a
2 multi-line string"""
```

Parameterized strings:

```
1 x = 12
2 print(f'x is {x}')    # prints "x is 12"
```

Python is generally *strongly typed* so it will often complain if you try to mix and match types. You can coerce a type with the `int()`, `float()`, `str()`, and `bool()` functions.

```
1 "2" + 4      # ERROR: can't mix types
2 int("2") + 4 # integer 6
3 "2" + str(4) # string 24
```

Arrays/Lists

JavaScript

In JS, lists are called *arrays*.

Arrays are zero-based.

Creating lists:

```
1 let a1 = new Array();    // Empty array
2 let a2 = new Array(10);  // Array of 10 elements
3 let a3 = [];            // Empty array
4 let a4 = [10, 20, 30]; // Array of 3 elements
5 let a5 = [1, 2, "b"];  // No problem
```

Accessing:

```
1 console.log(a4[1]); // prints 20
2
3 a4[0] = 5;    // change from 10 to 5
4 a4[20] = 99; // OK, makes a new element at index 20
```

Length/number of elements:

```
a4.length; // 3
```

Python

In Python, arrays are called *lists*.

Lists are zero-based.

Creating lists:

```
1 a1 = list()      # Empty list
2 a2 = list((88, 99)) # List of two elements
3 a3 = []          # Empty list
4 a4 = [10, 20, 30] # List of 3 elements
5 a5 = [1, 2, "b"]  # No problem
```

Accessing:

```
1 print(a4[1]) # prints 20
2
3 a4[0] = 5;    # change from 10 to 5
4 a4[20] = 99; # ERROR: assignment out of range
```

Length/Number of elements:

```
len(a4) # 3
```

Slices

In Python, we can access parts of lists or strings using slices.

Creating slices:

```
1 a[start:end] # items start through end-1
2 a[start:]     # items start through the rest of the array
3 a[:end]       # items from the beginning through end-1
4 a[:]          # a copy of the whole array
```

Starting from the end: We can also use negative numbers when creating slices, which just means we start with the index at the end of the array, rather than the index at the beginning of the array.

```
1 a[-1]      # last item in the array
2 a[-2:]     # last two items in the array
3 a[:-2]     # everything except the last two items
```

Tuples

Python supports a read-only type of list called a *tuple*.

```
1 x = (1, 2, 3)
2 print(x[1]) # prints 2
3
4 y = (10,)    # A tuple of one element, comma required
```

List Comprehensions

Python supports building lists with *list comprehensions*. This is often useful for filtering lists.

```
1 a = [1, 2, 3, 4, 5]
2
3 # Make a list b that is the same as list a:
4 b = [i for i in a] # Pretty boring
5
6 # Make a list c that contains only the
7 # even elements of a:
8 c = [i for i in a if i % 2 == 0]
```

Objects/Dicts

JavaScript

Objects hold data which can be found by a specific key called a *property*.

Creation:

```
1 let o1 = {};           // empty object
2 let o2 = {"x": 12};    // one property
3 let o3 = {y: "hello"}; // property quotes optional
4
5 let o4 = { // common multiline format
6   "a": 20,
7   "b": 1.2,
8   "foo": "hello"
9 };
```

Access:

```
1 console.log(o2.x);    // prints 12
2 console.log(o4["foo"]); // prints hello
```

Python

Dicts hold information that can be accessed by a *key*.

Unlike objects in JS, a `dict` is its own beast, and is not the same as an object obtained by instantiating a Python class.

Creation:

```

1 o1 = {}           # empty dict
2 o2 = {"x": 12}    # one key
3 o3 = {y: "hello"} # ERROR: key quotes required,
4                      # unless y is a variable
5                      # that holds a value you wish
6                      # to use as a key
7
8 o4 = { # multiline format
9     "a": 20,
10    "b": 1.2,
11    "foo": "hello"
12 }

```

Access:

```
print(o4["a"]) # Prints 20
```

Dot notation does not work with Python dicts.

String Formatting

JavaScript

Converting to different number bases:

```

1 let x = 237;
2 let x_binary = x.toString(2); // string '11101101'
3 let x_hex = x.toString(16);  // string 'ed'

```

Controlling floating point precision:

```

1 let x = 3.1415926535;
2 let y = x.toFixed(2); // string '3.14'

```

Padding and justification:

```

1 let s = "Hello!";
2 let t = s.padStart(10, ' '); // string '    Hello!'
3 let u = s.padEnd(10, ' '); // string 'Hello!    '
4
5 let v = s.padStart(10, '*'); // string '****Hello!'
6
7 // Pad with leading zeroes
8 (12).toString(2).padStart(8, '0'); // string '00001100'

```

Parameterized strings:

```

1 let x = 3.1415926;
2 let y = "Hello";
3 let z = 67;
4
5 // 'x is 3.14, y is "Hello", z is 01000011'

```

```
6 let s = `x is ${x.toFixed(2)}, y is "${y}", z is ${z.toString(2).padStart(8, '0')}`
```

Python

Python has the printf operator `%` which is tremendously powerful. (If the operands to `%` are numbers, modulo is performed. If the left operand is a string, printf is performed.)

But even `%` is being superseded by the `format` function.

Tons of details at pyformat.info.

Also see [printf-style String Formatting](#) for a reference.

Booleans and Conditionals

JavaScript

Literal boolean values:

```
1 x = true;
2 y = false;
```

Boolean operators:

Operator	Definition
<code>==</code>	Equality
<code>!=</code>	Inequality
<code>===</code>	Strict equality
<code>!==</code>	Strict inequality
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal

The concept of strict equality/inequality applies to items that might normally be converted into a compatible type. The strict tests will consider if the types themselves are the same.

```
1 0 == "0"; // true
2 0 === "0"; // false
3
4 0 == []; // true
5 0 === []; // false
6
7 0 == 0; // true
8 0 === 0; // true
```

Logical operators:

Operator	Description
!	Logical inverse, not
&&	Logical AND
	Logical OR

The not operator `!` can be used to test whether or not a value is "truthy".

```
1  !0;      // true
2  !!0;     // false
3  !!;       // false
4  !null;   // true
5  !"0";    // false, perhaps unexpectedly
6  !"x";    // false
```

Example:

```
1  if (a == 2 && b !== "") {
2      // Something complicated
3  }
```

Python

Literal boolean values:

```
1  x = True
2  y = False
```

Boolean operators:

Operator	Definition
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Logical operators:

Operator	Description
not	Logical inverse, not

and

Logical AND

or

Logical OR

The `not` operator can be used to test whether or not a value is "truthy".

```
1 not 0      # true
2 not not 0  # false
3 not 1      # false
4 not None;  # true
5 not "0"    # false, perhaps unexpectedly
6 not "x"    # false
```

Example:

```
1 if a == 2 and b != "":
2   # Something complicated
```

for Loops

JavaScript

C-style `for` loops:

```
1 for (let i = 0; i < 10; i++) {
2   console.log(i);
3 }
```

`for - in` loops iterate over the properties of an object or indexes of an array:

```
1 a = [10, 20, 30];
2
3 for (let i in a) {
4   console.log(i);    # 0 1 2
5   console.log(a[i]); # 10 20 30
6 }
7
8 b = {'x': 77, 'y': 88, 'z': 99};
9
10 for (let i in b) {
11   console.log(i);    # x y z
12   console.log(b[i]); # 77 88 99
13 }
```

`for - of` loops access the values within the array (as opposed to the indexes of the array):

```
1 a = [10, 20, 30];
2
3 for (let i of a) {
4   console.log(i);  # 10 20 30
5 }
```

Python

`for - in` loops over an *iterable*. This can be a list, object, or other type of iterable item.

Counting loops:

```
1 # Use the range() function to count:  
2 for i in range(10):  
3     print(i)    # Prints 0-9  
4  
5 for i in range(20, 30):  
6     print(i)    # Prints 20-29  
7  
8 for i in range(-10, 20, 3):  
9     print(i)    # Print every 3rd number from -10 to 19
```

Iterating over other types:

```
1 # A list  
2 a = [10, 20, 30]  
3  
4 # Print 10 20 30  
5 for i in a:  
6     print(i)  
7  
8 # A dict  
9 b = {'x':5, 'y':15, 'z':0}  
10  
11 # Print x y z (the keys of the dict)  
12 for i in b:  
13     print(i)
```

while Loops

JavaScript

C-style `while` and `do - while`:

```
1 // Print 10 down to 0:  
2  
3 let x = 10;  
4 while (x >= 0) {  
5     console.log(x);  
6     x--;  
7 }  
8  
9 // Print 0 up to 9:  
10 let x = 0;  
11 do {  
12     console.log(x);  
13     x++;  
14 } while (x < 10);
```

Python

Python has a `while` loop:

```
1 # Print from 10 down through 0
```

```
2 x = 10
3 while x >= 0:
4     print(x)
5     x -= 1
```

switch Statement

JavaScript

JS can switch on various data types:

```
1 switch(x) {
2     case "foo":
3         console.log("x is foo, all right");
4         break;
5     case 23:
6         console.log("but here x is 23");
7         break;
8     default:
9         console.log("x is something else entirely");
10 }
```

Python

Python doesn't have a `switch` statement. You can use `if - elif - else` blocks.

A somewhat clumsy approximation of a `switch` can be constructed with a `dict` of functions.

```
1 def func1():
2     print("case 1 is hit")
3
4 def func2():
5     print("case 2 is hit")
6
7 def func3():
8     print("case 3 is hit")
9
10 funcs = {
11     "alpha": func1,
12     "bravo": func2,
13     "charlie": func3
14 };
15
16 x = "bravo"
17 funcs[x]() # calls func2
```

if Conditionals

JavaScript

JS uses C-style `if` statements:

```
1 if (x == 10) {
2     console.log("x is 10");
3 } else if (x == 20) {
4     console.log("x is 20");
5 } else {
6     console.log("x is something else");
7 }
```

Python

Python notably uses `elif` instead of `else if`.

```
1 if x == 10:
2     print("x is 10")
3 elif x == 20:
4     print("x is 20")
5 else:
6     print("x is something else")
```

Classes

JavaScript

The current object is referred to by `this`.

Pre ES-2015, classes were created using functions. This is now outdated.

```
1 function Goat(color) {
2     this.legs = 4;
3     this.color = color;
4 }
5 g = new Goat("brown");
```

JS uses prototypal inheritance. Pre ES-2015, this was explicit, and is also outdated:

```
1 function Creature(type) {
2     this.type = type;
3 }
4
5 // Make Goats inherit from Creature:
6 Goat.prototype = new Creature("mammal");
7
8 // Add a method:
9 Goat.prototype.jump = function () {
10     console.log("I'm jumping! Yay!");
11 };
12
13 g = new Goat("red");
14 g.type; // "mammal", since Goat inherits from Creature
15 g.jump(); // "I'm jumping! Yay!"
```

Modern JS introduced the `class` keyword and a syntax more familiar to most other OOP languages. Note that the inheritance model is still prototypal inheritance; it's just that the details are hidden from the developer.

```
1 class Creature {
2     constructor(type) {
3         this.type = type;
4     }
5 }
6
7 class Goat extends Creature {
8     constructor(color) {
```

```

9     super("mammal");
10    this.legs = 4;
11    this.color = color;
12 }
13
14 jump() {
15   console.log("I'm jumping! Yay!");
16 }
17 }
18
19 g = new Goat("orange");
20 g.type; // "mammal"
21 g.jump(); // "I'm jumping! Yay!"

```

JS does not support multiple inheritance since each object can only have one prototype object. You have to use a *mix-in* if you want to achieve similar functionality.

Python

The current object is referred to by `self`. Note that `self` is explicitly present as the first parameter in object methods.

Python 2 syntax:

```

1 class Creature:
2   def __init__(self, type): # constructor
3     self.type = type
4
5 class Goat(Creature):
6   def __init__(self, color):
7     # call super constructor
8     Creature.__init__(self, "mammal")
9     self.color = color
10
11  def jump(self):
12    print("I'm jumping! Yay!")
13
14 g = Goat("green")
15 g.type # mammal
16 g.jump() # I'm jumping! Yay!

```

Python 3 syntax includes the new `super()` keyword to make life easier.

```

1 class Creature:
2   def __init__(self, type): # constructor
3     self.type = type
4
5 class Goat(Creature):
6   def __init__(self, color):
7     # call super constructor
8     super().__init__("mammal") # <-- Nicer!
9     self.color = color
10
11  def jump(self):
12    print("I'm jumping! Yay!")
13
14 g = Goat("green")
15 g.type # mammal
16 g.jump() # I'm jumping! Yay!

```

Python supports multiple inheritance.

Python Modules & Python Packages

This article explores Python **modules** and Python **packages**, two mechanisms that facilitate **modular programming**.

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to **modularizing** code in a large application:

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the [Zen of Python](#) is *Namespaces are one honking great idea—let's do more of those!*)

Functions, modules and packages are all constructs in Python that promote code modularization.

[Free PDF Download: Python 3 Cheat Sheet](#)

Python Modules: Overview

There are actually three different ways to define a **module** in Python:

1. A module can be written in Python itself.
2. A module can be written in C and loaded dynamically at run-time, like the `re` (**regular expression**) module.
3. A **built-in** module is intrinsically contained in the interpreter, like the `itertools` module.

A module's contents are accessed the same way in all three cases: with the `import` statement.

Here, the focus will mostly be on modules that are written in Python. The cool thing about modules written in Python is that they are exceedingly straightforward to build. All you need to do is create a file that contains legitimate Python code and then give the file a name with a `.py` extension. That's it! No special syntax or voodoo is necessary.

For example, suppose you have created a file called `mod.py` containing the following:

`mod.py`

```
1 s = "If Comrade Napoleon says it, it must be right."
2 a = [100, 200, 300]
3
4 def foo(arg):
5     print(f'arg = {arg}')
6
7 class Foo:
8     pass
```

Several objects are defined in `mod.py`:

- `s` (a string)
- `a` (a list)
- `foo()` (a function)
- `Foo` (a class)

Assuming `mod.py` is in an appropriate location, which you will learn more about shortly, these objects can be accessed by **importing** the module as follows:>>>

```

1 >>> import mod
2 >>> print(mod.s)
3 If Comrade Napoleon says it, it must be right.
4 >>> mod.a
5 [100, 200, 300]
6 >>> mod.foo(['quux', 'corge', 'grault'])
7 arg = ['quux', 'corge', 'grault']
8 >>> x = mod.Foo()
9 >>> x
10 <mod.Foo object at 0x03C181F0>

```

[Remove ads](#)

The Module Search Path

Continuing with the above example, let's take a look at what happens when Python executes the statement:

```
import mod
```

When the interpreter executes the above `import` statement, it searches for `mod.py` in a **list** of directories assembled from the following sources:

- The directory from which the input script was run or the **current directory** if the interpreter is being run interactively
- The list of directories contained in the `PYTHONPATH` environment variable, if it is set. (The format for `PYTHONPATH` is OS-dependent but should mimic the `PATH` environment variable.)
- An installation-dependent list of directories configured at the time Python is installed

The resulting search path is accessible in the Python variable `sys.path`, which is obtained from a module named `sys`:>>>

```

1 >>> import sys
2 >>> sys.path
3 ['', 'C:\\Users\\john\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',
4 'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
5 'C:\\Python36', 'C:\\Python36\\lib\\site-packages']

```

Note: The exact contents of `sys.path` are installation-dependent. The above will almost certainly look slightly different on your computer.

Thus, to ensure your module is found, you need to do one of the following:

- Put `mod.py` in the directory where the input script is located or the **current directory**, if interactive
- Modify the `PYTHONPATH` environment variable to contain the directory where `mod.py` is located before starting the interpreter
 - **Or:** Put `mod.py` in one of the directories already contained in the `PYTHONPATH` variable
- Put `mod.py` in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS

There is actually one additional option: you can put the module file in any directory of your choice and then modify `sys.path` at run-time so that it contains that directory. For example, in this case, you could put `mod.py` in directory `C:\Users\john` and then issue the following statements:>>>

```
1 >>> sys.path.append(r'C:\Users\john')
2 >>> sys.path
3 ['', 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc', 'C:\\\\Python36\\\\Lib\\\\idlelib',
4 'C:\\\\Python36\\\\python36.zip', 'C:\\\\Python36\\\\DLLs', 'C:\\\\Python36\\\\lib',
5 'C:\\\\Python36', 'C:\\\\Python36\\\\lib\\\\site-packages', 'C:\\\\Users\\\\john']
6 >>> import mod
```

Once a module has been imported, you can determine the location where it was found with the module's `__file__` attribute:>>>

```
1 >>> import mod
2 >>> mod.__file__
3 'C:\\\\Users\\\\john\\\\mod.py'
4
5 >>> import re
6 >>> re.__file__
7 'C:\\\\Python36\\\\lib\\\\re.py'
```

The directory portion of `__file__` should be one of the directories in `sys.path`.

The `import` Statement

Module contents are made available to the caller with the `import` statement. The `import` statement takes many different forms, shown below.

```
import <module_name>
```

The simplest form is the one already shown above:

```
import <module_name>
```

Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

The statement `import <module_name>` only places `<module_name>` in the caller's symbol table. The *objects* that are defined in the module *remain in the module's private symbol table*.

From the caller, objects in the module are only accessible when prefixed with `<module_name>` via **dot notation**, as illustrated below.

After the following `import` statement, `mod` is placed into the local symbol table. Thus, `mod` has meaning in the caller's local context:>>>

```
1 >>> import mod
2 >>> mod
3 <module 'mod' from 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc\\\\mod.py'>
```

But `s` and `foo` remain in the module's private symbol table and are not meaningful in the local context:>>>

```
1 >>> s
2 NameError: name 's' is not defined
3 >>> foo('quux')
4 NameError: name 'foo' is not defined
```

To be accessed in the local context, names of objects defined in the module must be prefixed by `mod` >>>

```
1 >>> mod.s
2 'If Comrade Napoleon says it, it must be right.'
3 >>> mod.foo('quux')
4 arg = quux
```

Several comma-separated modules may be specified in a single `import` statement:

```
import <module_name>[, <module_name> ...]
```

[Remove ads](#)

```
from <module_name> import <name(s)>
```

An alternate form of the `import` statement allows individual objects from the module to be imported *directly into the caller's symbol table*:

```
from <module_name> import <name(s)>
```

Following execution of the above statement, `<name(s)>` can be referenced in the caller's environment without the `<module_name>` prefix:>>>

```
1 >>> from mod import s, foo
2 >>> s
3 'If Comrade Napoleon says it, it must be right.'
4 >>> foo('quux')
5 arg = quux
6
7 >>> from mod import Foo
8 >>> x = Foo()
9 >>> x
10 <mod.Foo object at 0x02E3AD50>
```

Because this form of `import` places the object names directly into the caller's symbol table, any objects that already exist with the same name will be *overwritten*:>>>

```
1 >>> a = ['foo', 'bar', 'baz']
2 >>> a
3 ['foo', 'bar', 'baz']
4
5 >>> from mod import a
6 >>> a
7 [100, 200, 300]
```

It is even possible to indiscriminately `import` everything from a module at one fell swoop:

```
from <module_name> import *
```

This will place the names of *all* objects from `<module_name>` into the local symbol table, with the exception of any that begin with the underscore (`_`) character.

For example:>>>

```
1  >>> from mod import *
2  >>> s
3  'If Comrade Napoleon says it, it must be right.'
4  >>> a
5  [100, 200, 300]
6  >>> foo
7  <function foo at 0x03B449C0>
8  >>> Foo
9  <class 'mod.Foo'>
```

This isn't necessarily recommended in large-scale production code. It's a bit dangerous because you are entering names into the local symbol table *en masse*. Unless you know them all well and can be confident there won't be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

```
from <module_name> import <name> as <alt_name>
```

It is also possible to `import` individual objects but enter them into the local symbol table with alternate names:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> ...]
```

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:>>>

```
1  >>> s = 'foo'
2  >>> a = ['foo', 'bar', 'baz']
3
4  >>> from mod import s as string, a as alist
5  >>> s
6  'foo'
7  >>> string
8  'If Comrade Napoleon says it, it must be right.'
9  >>> a
10 ['foo', 'bar', 'baz']
11 >>> alist
12 [100, 200, 300]
```

```
import <module_name> as <alt_name>
```

You can also import an entire module under an alternate name:

```
import <module_name> as <alt_name>
```

```
>>>
```

```
1 >>> import mod as my_module
2 >>> my_module.a
3 [100, 200, 300]
4 >>> my_module.foo('qux')
5 arg = qux
```

Module contents can be imported from within a [function definition](#). In that case, the `import` does not occur until the function is *called*:>>>

```
1 >>> def bar():
2 ...     from mod import foo
3 ...     foo('corge')
4 ...
5
6 >>> bar()
7 arg = corge
```

However, **Python 3** does not allow the indiscriminate `import *` syntax from within a function:>>>

```
1 >>> def bar():
2 ...     from mod import *
3 ...
4 SyntaxError: import * only allowed at module level
```

Lastly, a `try` statement with an `except ImportError` clause can be used to guard against unsuccessful `import` attempts:>>>

```
1 >>> try:
2 ...     # Non-existent module
3 ...     import baz
4 ... except ImportError:
5 ...     print('Module not found')
6 ...
7
8 Module not found
```

```
>>>
```

```
1 >>> try:
2 ...     # Existing module, but non-existent object
3 ...     from mod import baz
4 ... except ImportError:
5 ...     print('Object not found in module')
6 ...
7
8 Object not found in module
```

[Remove ads](#)

The `dir()` Function

The built-in function `dir()` returns a list of defined names in a namespace. Without arguments, it produces an alphabetically sorted list of names in the current **local symbol table**:>>>

```
1 >>> dir()
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> qux = [1, 2, 3, 4, 5]
6 >>> dir()
7 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
8 '__package__', '__spec__', 'qux']
9
10 >>> class Bar():
11     ...     pass
12 ...
13 >>> x = Bar()
14 >>> dir()
15 ['Bar', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
16 '__package__', '__spec__', 'qux', 'x']
```

Note how the first call to `dir()` above lists several names that are automatically defined and already in the namespace when the interpreter starts. As new names are defined (`qux`, `Bar`, `x`), they appear on subsequent invocations of `dir()`.

This can be useful for identifying what exactly has been added to the namespace by an import statement:>>>

```
1 >>> dir()
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> import mod
6 >>> dir()
7 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
8 '__package__', '__spec__', 'mod']
9 >>> mod.s
10 'If Comrade Napoleon says it, it must be right.'
11 >>> mod.foo([1, 2, 3])
12 arg = [1, 2, 3]
13
14 >>> from mod import a, Foo
15 >>> dir()
16 ['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
17 '__package__', '__spec__', 'a', 'mod']
18 >>> a
19 [100, 200, 300]
20 >>> x = Foo()
21 >>> x
22 <mod.Foo object at 0x002EAD50>
23
24 >>> from mod import s as string
25 >>> dir()
26 ['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
27 '__package__', '__spec__', 'a', 'mod', 'string', 'x']
28 >>> string
29 'If Comrade Napoleon says it, it must be right.'
```

When given an argument that is the name of a module, `dir()` lists the names defined in the module:>>>

```
1 >>> import mod
2 >>> dir(mod)
3 ['Foo', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
4 '__name__', '__package__', '__spec__', 'a', 'foo', 's']
```

```
>>>
```

```
1 >>> dir()
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4 >>> from mod import *
5 >>> dir()
6 ['__Foo__', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
7 '__package__', '__spec__', 'a', 'foo', 's']
```

Executing a Module as a Script

Any `.py` file that contains a **module** is essentially also a Python **script**, and there isn't any reason it can't be executed like one.

Here again is `mod.py` as it was defined above:

mod.py

```
1 s = "If Comrade Napoleon says it, it must be right."
2 a = [100, 200, 300]
3
4 def foo(arg):
5     print(f'arg = {arg}')
6
7 class Foo:
8     pass
```

This can be run as a script:

```
1 C:\Users\john\Documents>python mod.py
2 C:\Users\john\Documents>
```

There are no errors, so it apparently worked. Granted, it's not very interesting. As it is written, it only *defines* objects. It doesn't *do* anything with them, and it doesn't generate any output.

Let's modify the above Python module so it does generate some output when run as a script:

mod.py

```
1 s = "If Comrade Napoleon says it, it must be right."
2 a = [100, 200, 300]
3
4 def foo(arg):
5     print(f'arg = {arg}')
6
7 class Foo:
8     pass
9
10 print(s)
11 print(a)
12 foo('quux')
13 x = Foo()
14 print(x)
```

Now it should be a little more interesting:

```
1 C:\Users\john\Documents>python mod.py
2 If Comrade Napoleon says it, it must be right.
3 [100, 200, 300]
4 arg = quux
5 <__main__.Foo object at 0x02F101D0>
```

Unfortunately, now it also generates output when imported as a module:>>>

```
1 >>> import mod
2 If Comrade Napoleon says it, it must be right.
3 [100, 200, 300]
4 arg = quux
5 <mod.Foo object at 0x0169AD50>
```

This is probably not what you want. It isn't usual for a module to generate output when it is imported.

Wouldn't it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script?

Ask and ye shall receive.

When a `.py` file is imported as a module, Python sets the special **dunder** variable `__name__` to the name of the module. However, if a file is run as a standalone script, `__name__` is (creatively) set to the string `'__main__'`. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

mod.py

```
1 s = "If Comrade Napoleon says it, it must be right."
2 a = [100, 200, 300]
3
4 def foo(arg):
5     print(f'arg = {arg}')
6
7 class Foo:
8     pass
9
10 if (__name__ == '__main__'):
11     print('Executing as standalone script')
12     print(s)
13     print(a)
14     foo('quux')
15     x = Foo()
16     print(x)
```

Now, if you run as a script, you get output:

```
1 C:\Users\john\Documents>python mod.py
2 Executing as standalone script
3 If Comrade Napoleon says it, it must be right.
4 [100, 200, 300]
5 arg = quux
6 <__main__.Foo object at 0x03450690>
```

But if you import as a module, you don't:>>>

```
1 >>> import mod
```

```
2 >>> mod.foo('grault')
3 arg = grault
```

Modules are often designed with the capability to run as a standalone script for purposes of testing the functionality that is contained within the module. This is referred to as **unit testing**. For example, suppose you have created a module `fact.py` containing a **factorial** function, as follows:

fact.py

```
1 def fact(n):
2     return 1 if n == 1 else n * fact(n-1)
3
4 if (__name__ == '__main__'):
5     import sys
6     if len(sys.argv) > 1:
7         print(fact(int(sys.argv[1])))
```

The file can be treated as a module, and the `fact()` function imported:>>>

```
1 >>> from fact import fact
2 >>> fact(6)
3 720
```

But it can also be run as a standalone by passing an integer argument on the command-line for testing:

```
1 C:\Users\john\Documents>python fact.py 6
2 720
```

[Remove ads](#)

Reloading a Module

For reasons of efficiency, a module is only loaded once per interpreter session. That is fine for function and class definitions, which typically make up the bulk of a module's contents. But a module can contain executable statements as well, usually for initialization. Be aware that these statements will only be executed the *first time* a module is imported.

Consider the following file `mod.py` :

mod.py

```
1 a = [100, 200, 300]
2 print('a =', a)
```

>>>

```
1 >>> import mod
2 a = [100, 200, 300]
3 >>> import mod
4 >>> import mod
```

```
5
6 >>> mod.a
7 [100, 200, 300]
```

The `print()` statement is not executed on subsequent imports. (For that matter, neither is the assignment statement, but as the final display of the value of `mod.a` shows, that doesn't matter. Once the assignment is made, it sticks.)

If you make a change to a module and need to reload it, you need to either restart the interpreter or use a function called `reload()` from module `importlib`:

```
1 >>> import mod
2 a = [100, 200, 300]
3
4 >>> import mod
5
6 >>> import importlib
7 >>> importlib.reload(mod)
8 a = [100, 200, 300]
9 <module 'mod' from 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc\\\\mod.py'>
```

Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the



following arrangement:



Here, there is a directory named `pkg` that contains two modules, `mod1.py` and `mod2.py`. The contents of the modules are:

mod1.py

```
1 def foo():
2     print('[mod1] foo()')
3
4 class Foo:
5     pass
```

mod2.py

```
1 def bar():
2     print('[mod2] bar()')
3
4 class Bar:
5     pass
```

Given this structure, if the `pkg` directory resides in a location where it can be found (in one of the directories contained in `sys.path`), you can refer to the two **modules with dot notation** (`pkg.mod1`, `pkg.mod2`) and import them with the syntax you are already familiar with:

```
import <module_name>[, <module_name> ...]
```

```
>>>
```

```
1 >>> import pkg.mod1, pkg.mod2
2 >>> pkg.mod1.foo()
3 [mod1] foo()
4 >>> x = pkg.mod2.Bar()
5 >>> x
6 <pkg.mod2.Bar object at 0x033F7290>
```

```
from <module_name> import <name(s)>
```

```
>>>
```

```
1 >>> from pkg.mod1 import foo
2 >>> foo()
3 [mod1] foo()
```

```
from <module_name> import <name> as <alt_name>
```

```
>>>
```

```
1 >>> from pkg.mod2 import Bar as Qux
2 >>> x = Qux()
3 >>> x
4 <pkg.mod2.Bar object at 0x036DFFD0>
```

You can import modules with these statements as well:

```
1 from <package_name> import <modules_name>[, <module_name> ...]
2 from <package_name> import <module_name> as <alt_name>
```

```
>>>
```

```
1 >>> from pkg import mod1
2 >>> mod1.foo()
3 [mod1] foo()
4
5 >>> from pkg import mod2 as quux
6 >>> quux.bar()
```

```
7 [mod2] bar()
```

You can technically import the package as well:>>>

```
1 >>> import pkg
2 >>> pkg
3 <module 'pkg' (namespace)>
```

But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn't do much of anything useful. In particular, it *does not place* any of the modules in `pkg` into the local namespace:>>>

```
1 >>> pkg.mod1
2 Traceback (most recent call last):
3   File "<pyshell#34>", line 1, in <module>
4     pkg.mod1
5 AttributeError: module 'pkg' has no attribute 'mod1'
6 >>> pkg.mod1.foo()
7 Traceback (most recent call last):
8   File "<pyshell#35>", line 1, in <module>
9     pkg.mod1.foo()
10 AttributeError: module 'pkg' has no attribute 'mod1'
11 >>> pkg.mod2.Bar()
12 Traceback (most recent call last):
13   File "<pyshell#36>", line 1, in <module>
14     pkg.mod2.Bar()
15 AttributeError: module 'pkg' has no attribute 'mod2'
```

To actually import the modules or their contents, you need to use one of the forms shown above. [Remove ads](#)

Package Initialization

If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

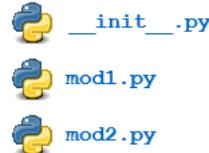
For example, consider the following `__init__.py` file:

`__init__.py`

```
1 print(f'Invoking __init__.py for {__name__}')
2 A = ['quux', 'corge', 'grault']
```



Let's add this file to the `pkg` directory from the above example:



Now when the package is imported, the global list `A` is initialized:>>>

```
1 >>> import pkg
2 Invoking __init__.py for pkg
3 >>> pkg.A
4 ['quux', 'corge', 'grault']
```

A **module** in the package can access the global variable by importing it in turn:

mod1.py

```
1 def foo():
2     from pkg import A
3     print('[mod1] foo() / A = ', A)
4
5 class Foo:
6     pass
```

>>>

```
1 >>> from pkg import mod1
2 Invoking __init__.py for pkg
3 >>> mod1.foo()
4 [mod1] foo() / A =  ['quux', 'corge', 'grault']
```

`__init__.py` can also be used to effect automatic importing of modules from a package. For example, earlier you saw that the statement `import pkg` only places the name `pkg` in the caller's local symbol table and doesn't import any modules. But if `__init__.py` in the `pkg` directory contains the following:

__init__.py

```
1 print(f'Invoking __init__.py for {__name__}')
2 import pkg.mod1, pkg.mod2
```

then when you execute `import pkg`, modules `mod1` and `mod2` are imported automatically:>>>

```
1 >>> import pkg
2 Invoking __init__.py for pkg
3 >>> pkg.mod1.foo()
4 [mod1] foo()
5 >>> pkg.mod2.bar()
6 [mod2] bar()
```

Note: Much of the Python documentation states that an `__init__.py` file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of `__init__.py` signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.

Starting with **Python 3.3**, **Implicit Namespace Packages** were introduced. These allow for the creation of a package without any `__init__.py` file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

Importing * From a Package

For the purposes of the following discussion, the previously defined package is expanded to contain some additional modules:



There are now four modules defined in the `pkg` directory. Their contents are as shown below:

`mod1.py`

```
1 def foo():
2     print('[mod1] foo()')
3
4 class Foo:
5     pass
```

`mod2.py`

```
1 def bar():
2     print('[mod2] bar()')
3
4 class Bar:
5     pass
```

`mod3.py`

```
1 def baz():
2     print('[mod3] baz()')
3
4 class Baz:
5     pass
```

`mod4.py`

```
1 def qux():
2     print('[mod4] qux()')
3
4 class Qux:
5     pass
```

(Imaginative, aren't they?)

You have already seen that when `import *` is used for a **module**, *all* objects from the module are imported into the local symbol table, except those whose names begin with an underscore, as always:>>>

```
1 >>> dir()
```

```
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> from pkg.mod3 import *
6
7 >>> dir()
8 ['Baz', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
9 '__package__', '__spec__', 'baz']
10 >>> baz()
11 [mod3] baz()
12 >>> Baz
13 <class 'pkg.mod3.Baz'>
```

The analogous statement for a **package** is this:

```
from <package_name> import *
```

What does that do?>>>

```
1 >>> dir()
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> from pkg import *
6 >>> dir()
7 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
8 '__package__', '__spec__']
```

Hmph. Not much. You might have expected (assuming you had any expectations at all) that Python would dive down into the package directory, find all the modules it could, and import them all. But as you can see, by default that is not what happens.

Instead, Python follows this convention: if the `__init__.py` file in the **package** directory contains a **list** named `__all__`, it is taken to be a list of modules that should be imported when the statement `from <package_name> import *` is encountered.

For the present example, suppose you create an `__init__.py` in the `pkg` directory like this:

`pkg/__init__.py`

```
1 __all__ = [
2     'mod1',
3     'mod2',
4     'mod3',
5     'mod4'
6 ]
```

Now `from pkg import *` imports all four modules:>>>

```
1 >>> dir()
2 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> from pkg import *
6 >>> dir()
7 ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
8 '__package__', '__spec__', 'mod1', 'mod2', 'mod3', 'mod4']
9 >>> mod2.bar()
```

```
10 [mod2] bar()
11 >>> mod4.Qux
12 <class 'pkg.mod4.Qux'>
```

Using `import *` still isn't considered terrific form, any more for **packages** than for **modules**. But this facility at least gives the creator of the package some control over what happens when `import *` is specified. (In fact, it provides the capability to disallow it entirely, simply by declining to define `__all__` at all. As you have seen, the default behavior for packages is to import nothing.)

By the way, `__all__` can be defined in a **module** as well and serves the same purpose: to control what is imported with `import *`. For example, modify `mod1.py` as follows:

`pkg/mod1.py`

```
1 __all__ = ['foo']
2
3 def foo():
4     print('[mod1] foo()')
5
6 class Foo:
7     pass
```

Now an `import *` statement from `pkg.mod1` will only import what is contained in `__all__`:>>>

```
1 >>> dir()
2['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
3 '__package__', '__spec__']
4
5 >>> from pkg.mod1 import *
6 >>> dir()
7['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
8 '__package__', '__spec__', 'foo']
9
10 >>> foo()
11 [mod1] foo()
12 >>> Foo
13 Traceback (most recent call last):
14   File "<pyshell#37>", line 1, in <module>
15     Foo
16 NameError: name 'Foo' is not defined
```

`foo()` (the function) is now defined in the local namespace, but `Foo` (the class) is not, because the latter is not in `__all__`.

In summary, `__all__` is used by both **packages** and **modules** to control what is imported when `import *` is specified. But *the default behavior differs*:

- For a package, when `__all__` is not defined, `import *` does not import anything.
- For a module, when `__all__` is not defined, `import *` imports everything (except—you guessed it—names starting with an underscore).

[Remove ads](#)

Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let's make one more modification to the example **package**



directory as follows:

The four modules (`mod1.py` , `mod2.py` , `mod3.py` and `mod4.py`) are defined as previously. But now, instead of being lumped together into the `pkg` directory, they are split out into two **subpackage** directories, `sub_pkg1` and `sub_pkg2` .

Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:>>>

```
1 >>> import pkg.sub_pkg1.mod1
2 >>> pkg.sub_pkg1.mod1.foo()
3 [mod1] foo()
4
5 >>> from pkg.sub_pkg1 import mod2
6 >>> mod2.bar()
7 [mod2] bar()
8
9 >>> from pkg.sub_pkg2.mod3 import baz
10 >>> baz()
11 [mod3] baz()
12
13 >>> from pkg.sub_pkg2.mod4 import qux as grault
14 >>> grault()
15 [mod4] qux()
```

In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need). For example, suppose you want to import and execute function `foo()` (defined in module `mod1`) from within module `mod3` . You can either use an **absolute import**:

pkg/sub_pkg2/mod3.py

```
1 def baz():
2     print('[mod3] baz()')
3
4 class Baz:
5     pass
6
7 from pkg.sub_pkg1.mod1 import foo
8 foo()
```

>>>

```
1 >>> from pkg.sub_pkg2 import mod3
2 [mod1] foo()
3 >>> mod3.foo()
4 [mod1] foo()
```

Or you can use a **relative import**, where `..` refers to the package one level up. From within `mod3.py`, which is in subpackage `sub_pkg2`,

- `..` evaluates to the parent package (`pkg`), and
- `..sub_pkg1` evaluates to subpackage `sub_pkg1` of the parent package.

`pkg/sub_pkg2/mod3.py`

```
1 def baz():
2     print('[mod3] baz()')
3
4 class Baz:
5     pass
6
7 from .. import sub_pkg1
8 print(sub_pkg1)
9
10 from ..sub_pkg1.mod1 import foo
11 foo()
```

>>>

```
1 >>> from pkg.sub_pkg2 import mod3
2 <module 'pkg.sub_pkg1' (namespace)>
3 [mod1] foo()
```

Conclusion

In this tutorial, you covered the following topics:

- How to create a Python **module**
- Locations where the Python interpreter searches for a module
- How to obtain access to the objects defined in a module with the `import` statement
- How to create a module that is executable as a standalone script
- How to organize modules into **packages** and **subpackages**
- How to control package initialization

Free PDF Download: Python 3 Cheat Sheet

This will hopefully allow you to better understand how to gain access to the functionality available in the many third-party and built-in modules available in Python.

Additionally, if you are developing your own application, creating your own **modules** and **packages** will help you organize and modularize your code, which makes coding, maintenance, and debugging easier.

If you want to learn more, check out the following documentation at [Python.org](https://www.python.org):

`__main__` – Top-level script environment

`'__main__'` is the name of the scope in which top-level code executes. A module's `__name__` is set equal to `'__main__'` when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run as a script or with `python -m` but not when it is imported:

```
1 if __name__ == "__main__":
2     # execute only if run as a script
3     main()
```

For a package, the same effect can be achieved by including a `__main__.py` module, the contents of which will be executed when the module is run

. The import system

Python code in one `module` gains access to the code in another module by the process of `importing` it. The `import` statement is the most common way of invoking the import machinery, but it is not the only way. Functions such as `importlib.import_module()` and built-in `__import__()` can also be used to invoke the import machinery.

The `import` statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope. The search operation of the `import` statement is defined as a call to the `__import__()` function, with the appropriate arguments. The return value of `__import__()` is used to perform the name binding operation of the `import` statement. See the `import` statement for the exact details of that name binding operation.

A direct call to `__import__()` performs only the module search and, if found, the module creation operation. While certain side-effects may occur, such as the importing of parent packages, and the updating of various caches (including `sys.modules`), only the `import` statement performs a name binding operation.

When an `import` statement is executed, the standard builtin `__import__()` function is called. Other mechanisms for invoking the import system (such as `importlib.import_module()`) may choose to bypass `__import__()` and use their own solutions to implement import semantics.

When a module is first imported, Python searches for the module and if found, it creates a module object 1, initializing it. If the named module cannot be found, a `ModuleNotFoundError` is raised. Python implements various strategies to search for the named module when the import machinery is invoked. These strategies can be modified and extended by using various hooks described in the sections below.

Changed in version 3.3: The import system has been updated to fully implement the second phase of [PEP 302](#). There is no longer any implicit import machinery - the full import system is exposed through `sys.meta_path`. In addition, native namespace package support has been implemented (see [PEP 420](#)).

5.1. `importlib`

The `importlib` module provides a rich API for interacting with the import system. For example `importlib.import_module()` provides a recommended, simpler API than built-in `__import__()` for invoking the import machinery. Refer to the `importlib` library documentation for additional detail.

5.2. Packages

Python has only one type of module object, and all modules are of this type, regardless of whether the module is implemented in Python, C, or something else. To help organize modules and provide a naming hierarchy, Python has a concept of [packages](#).

You can think of packages as the directories on a file system and modules as files within directories, but don't take this analogy too literally since packages and modules need not originate from the file system. For the purposes of this documentation, we'll use this convenient analogy of directories and files. Like file system directories, packages are organized hierarchically, and packages may themselves contain subpackages, as well as regular modules.

It's important to keep in mind that all packages are modules, but not all modules are packages. Or put another way, packages are just a special kind of module. Specifically, any module that contains a `__path__` attribute is considered a package.

All modules have a name. Subpackage names are separated from their parent package name by a dot, akin to Python's standard attribute access syntax. Thus you might have a module called `sys` and a package called `email`, which in turn has a subpackage called `email.mime` and a module within that subpackage called `email.mime.text`.

5.2.1. Regular packages

Python defines two types of packages, [regular packages](#) and [namespace packages](#). Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an `__init__.py` file. When a regular package is imported, this `__init__.py` file is implicitly executed, and the objects it defines are bound to names in the package's namespace. The `__init__.py` file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

For example, the following file system layout defines a top level `parent` package with three subpackages:

```
1 parent/
2     __init__.py
3     one/
4         __init__.py
5     two/
6         __init__.py
7     three/
8         __init__.py
```

Importing `parent.one` will implicitly execute `parent/__init__.py` and `parent/one/__init__.py`. Subsequent imports of `parent.two` or `parent.three` will execute `parent/two/__init__.py` and `parent/three/__init__.py` respectively.

5.2.2. Namespace packages

A namespace package is a composite of various [portions](#), where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.

Namespace packages do not use an ordinary list for their `__path__` attribute. They instead use a custom iterable type which will automatically perform a new search for package portions on the next import attempt within that package if the path of their parent package (or `sys.path` for a top level package) changes.

With namespace packages, there is no `parent/__init__.py` file. In fact, there may be multiple `parent` directories found during import search, where each one is provided by a different portion. Thus `parent/one` may not be physically located next to `parent/two`. In this case, Python will create a namespace package for the top-level `parent` package whenever it or one of its subpackages is imported.

See also [PEP 420](#) for the namespace package specification.

5.3. Searching

To begin the search, Python needs the [fully qualified](#) name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the `import` statement, or from the parameters to the `importlib.import_module()` or `__import__()` functions.

This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a `ModuleNotFoundError` is raised.

5.3.1. The module cache

The first place checked during import search is `sys.modules`. This mapping serves as a cache of all modules that have been previously imported, including the intermediate paths. So if `foo.bar.baz` was previously imported, `sys.modules` will contain entries for `foo`, `foo.bar`, and `foo.bar.baz`. Each key will have as its value the corresponding module object.

During import, the module name is looked up in `sys.modules` and if present, the associated value is the module satisfying the import, and the process completes. However, if the value is `None`, then a `ModuleNotFoundError` is raised. If the module name is missing, Python will continue searching for the module.

`sys.modules` is writable. Deleting a key may not destroy the associated module (as other modules may hold references to it), but it will invalidate the cache entry for the named module, causing Python to search anew for the named module upon its next import. The key can also be assigned to `None`, forcing the next import of the module to result in a `ModuleNotFoundError`.

Beware though, as if you keep a reference to the module object, invalidate its cache entry in `sys.modules`, and then re-import the named module, the two module objects will *not* be the same. By contrast, `importlib.reload()` will reuse the *same* module object, and simply reinitialise the module contents by rerunning the module's code.

5.3.2. Finders and loaders

If the named module is not found in `sys.modules`, then Python's import protocol is invoked to find and load the module. This protocol consists of two conceptual objects, [finders](#) and [loaders](#). A finder's job is to determine whether it can find the named module using whatever strategy it knows about. Objects that implement both of these interfaces are referred to as [importers](#) - they return themselves when they find that they can load the requested module.

Python includes a number of default finders and importers. The first one knows how to locate built-in modules, and the second knows how to locate frozen modules. A third default finder searches an [import path](#) for modules. The [import path](#) is a list of locations that may name file system paths or zip files. It can also be extended to search for any locatable resource, such as those identified by URLs.

The import machinery is extensible, so new finders can be added to extend the range and scope of module searching.

Finders do not actually load modules. If they can find the named module, they return a *module spec*, an encapsulation of the module's import-related information, which the import machinery then uses when loading the module.

The following sections describe the protocol for finders and loaders in more detail, including how you can create and register new ones to extend the import machinery.

Changed in version 3.4: In previous versions of Python, finders returned `loaders` directly, whereas now they return module specs which *contain* loaders. Loaders are still used during import but have fewer responsibilities.

5.3.3. Import hooks

The import machinery is designed to be extensible; the primary mechanism for this are the *import hooks*. There are two types of import hooks: *meta hooks* and *import path hooks*.

Meta hooks are called at the start of import processing, before any other import processing has occurred, other than `sys.modules` cache look up. This allows meta hooks to override `sys.path` processing, frozen modules, or even built-in modules. Meta hooks are registered by adding new finder objects to `sys.meta_path`, as described below.

Import path hooks are called as part of `sys.path` (or `package.__path__`) processing, at the point where their associated path item is encountered. Import path hooks are registered by adding new callables to `sys.path_hooks` as described below.

5.3.4. The meta path

When the named module is not found in `sys.modules`, Python next searches `sys.meta_path`, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called `find_spec()` which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

If the meta path finder knows how to handle the named module, it returns a spec object. If it cannot handle the named module, it returns `None`. If `sys.meta_path` processing reaches the end of its list without returning a spec, then a `ModuleNotFoundError` is raised. Any other exceptions raised are simply propagated up, aborting the import process.

The `find_spec()` method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example `foo.bar.baz`. The second argument is the path entries to use for the module search. For top-level modules, the second argument is `None`, but for submodules or subpackages, the second argument is the value of the parent package's `__path__` attribute. If the appropriate `__path__` attribute cannot be accessed, a `ModuleNotFoundError` is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

The meta path may be traversed multiple times for a single import request. For example, assuming none of the modules involved has already been cached, importing `foo.bar.baz` will first perform a top level import, calling `mpf.find_spec("foo", None, None)` on each meta path finder (`mpf`). After `foo` has been imported, `foo.bar` will be imported by traversing the meta path a second time, calling `mpf.find_spec("foo.bar", foo.__path__, None)`. Once `foo.bar` has been imported, the final traversal will call `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Some meta path finders only support top level imports. These importers will always return `None` when anything other than `None` is passed as the second argument.

Python's default `sys.meta_path` has three meta path finders, one that knows how to import built-in modules, one that knows how to import frozen modules, and one that knows how to import modules from an *import path* (i.e. the *path based finder*).

Changed in version 3.4: The `find_spec()` method of meta path finders replaced `find_module()`, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement `find_spec()`.

5.4. Loading

If and when a module spec is found, the import machinery will use it (and the loader it contains) when loading the module. Here is an approximation of what happens during the loading portion of import:

```

1 module = None
2 if spec.loader is not None and hasattr(spec.loader, 'create_module'):
3     # It is assumed 'exec_module' will also be defined on the loader.
4     module = spec.loader.create_module(spec)
5 if module is None:
6     module = ModuleType(spec.name)
7 # The import-related module attributes get set here:
8 _init_module_attrs(spec, module)
9
10 if spec.loader is None:
11     # unsupported
12     raise ImportError
13 if spec.origin is None and spec.submodule_search_locations is not None:
14     # namespace package
15     sys.modules[spec.name] = module
16 elif not hasattr(spec.loader, 'exec_module'):
17     module = spec.loader.load_module(spec.name)
18     # Set __loader__ and __package__ if missing.
19 else:
20     sys.modules[spec.name] = module
21     try:
22         spec.loader.exec_module(module)
23     except BaseException:
24         try:
25             del sys.modules[spec.name]
26         except KeyError:
27             pass
28     raise
29 return sys.modules[spec.name]

```

Note the following details:

- If there is an existing module object with the given name in `sys.modules`, import will have already returned it.
- The module will exist in `sys.modules` before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to `sys.modules` beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module – and only the failing module – gets removed from `sys.modules`. Any module already in the `sys.modules` cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in `sys.modules`.
- After the module is created but before execution, the import machinery sets the import-related module attributes (“`_init_module_attrs`” in the pseudo-code example above), as summarized in a [later section](#).
- Module execution is the key moment of loading in which the module’s namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to `exec_module()` may not be the one returned at the end of import 2.

Changed in version 3.4: The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the `importlib.abc.Loader.load_module()` method.

5.4.1. Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module’s code in the module’s global name space (`module.__dict__`).
- If the loader cannot execute the module, it should raise an `ImportError`, although any other exception raised during `exec_module()` will be propagated.

In many cases, the finder and loader can be the same object; in such cases the `find_spec()` method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a `create_module()` method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

New in version 3.4: The `create_module()` method of loaders.

Changed in version 3.4: The `load_module()` method was replaced by `exec_module()` and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.
- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Changed in version 3.5: A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

Changed in version 3.6: An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.

5.4.2. Submodules

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module's namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let's say you have the following directory structure:

```
1 spam/
2     __init__.py
3     foo.py
4     bar.py
```

and `spam/__init__.py` has the following lines in it:

```
1 from .foo import Foo
2 from .bar import Bar
```

then executing the following puts a name binding to `foo` and `bar` in the `spam` module:>>>

```
1 >>> import spam
```

```
2 >>> spam.foo
3 <module 'spam.foo' from '/tmp/imports/spam/foo.py'>
4 >>> spam.bar
5 <module 'spam.bar' from '/tmp/imports/spam/bar.py'>
```

Given Python's familiar name binding rules this might seem surprising, but it's actually a fundamental feature of the import system. The invariant holding is that if you have `sys.modules['spam']` and `sys.modules['spam.foo']` (as you would after the above import), the latter must appear as the `foo` attribute of the former.

5.4.3. Module spec

The import machinery uses a variety of information about each module during import, especially before loading. Most of the information is common to all modules. The purpose of a module's spec is to encapsulate this import-related information on a per-module basis.

Using a spec during import allows state to be transferred between import system components, e.g. between the finder that creates the module spec and the loader that executes it. Most importantly, it allows the import machinery to perform the boilerplate operations of loading, whereas without a module spec the loader had that responsibility.

The module's spec is exposed as the `__spec__` attribute on a module object. See [ModuleSpec](#) for details on the contents of the module spec.

New in version 3.4.

5.4.4. Import-related module attributes

The import machinery fills in these attributes on each module object during loading, based on the module's spec, before the loader executes the module. `__name__`

The `__name__` attribute must be set to the fully-qualified name of the module. This name is used to uniquely identify the module in the import system. `__loader__`

The `__loader__` attribute must be set to the loader object that the import machinery used when loading the module. This is mostly for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

`__package__`

The module's `__package__` attribute must be set. Its value must be a string, but it can be the same value as its `__name__`. When the module is a package, its `__package__` value should be set to its `__name__`. When the module is not a package, `__package__` should be set to the empty string for top-level modules, or for submodules, to the parent package's name. See [PEP 366](#) for further details.

This attribute is used instead of `__name__` to calculate explicit relative imports for main modules, as defined in [PEP 366](#). It is expected to have the same value as `__spec__.parent`.

Changed in version 3.6: The value of `__package__` is expected to be the same as `__spec__.parent`.

The `__spec__` attribute must be set to the module spec that was used when importing the module. Setting `__spec__` appropriately applies equally to [modules initialized during interpreter startup](#). The one exception is `__main__`, where `__spec__` is set to [None](#) in some cases.

When `__package__` is not defined, `__spec__.parent` is used as a fallback.

New in version 3.4.

Changed in version 3.6: `__spec__.parent` is used as a fallback when `__package__` is not defined. `__path__`

If the module is a package (either regular or namespace), the module object's `__path__` attribute must be set. The value must be iterable, but may be empty if `__path__` has no further significance. If `__path__` is not empty, it must produce strings when iterated

over. More details on the semantics of `__path__` are given [below](#).

Non-package modules should not have a `__path__` attribute. `__file__`, `__cached__`

`__file__` is optional. If set, this attribute's value must be a string. The import system may opt to leave `__file__` unset if it has no semantic meaning (e.g. a module loaded from a database).

If `__file__` is set, it may also be appropriate to set the `__cached__` attribute which is the path to any compiled version of the code (e.g. byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file would exist (see [PEP 3147](#)).

It is also appropriate to set `__cached__` when `__file__` is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of `__file__` and/or `__cached__`. So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

5.4.5. module.`__path__`

By definition, if a module has a `__path__` attribute, it is a package.

A package's `__path__` attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as `sys.path`, i.e. providing a list of locations to search for modules during import. However, `__path__` is typically much more constrained than `sys.path`.

`__path__` must be an iterable of strings, but it may be empty. The same rules used for `sys.path` also apply to a package's `__path__`, and `sys.path_hooks` (described below) are consulted when traversing a package's `__path__`.

A package's `__init__.py` file may set or alter the package's `__path__` attribute, and this was typically the way namespace packages were implemented prior to [PEP 420](#). With the adoption of [PEP 420](#), namespace packages no longer need to supply `__init__.py` files containing only `__path__` manipulation code; the import machinery automatically sets `__path__` correctly for the namespace package.

5.4.6. Module reprs

By default, all modules have a usable repr, however depending on the attributes set above, and in the module's spec, you can more explicitly control the repr of module objects.

If the module has a spec (`__spec__`), the import machinery will try to generate a repr from it. If that fails or there is no spec, the import system will craft a default repr using whatever information is available on the module. It will try to use the `module.__name__`, `module.__file__`, and `module.__loader__` as input into the repr, with defaults for whatever information is missing.

Here are the exact rules used:

- If the module has a `__spec__` attribute, the information in the spec is used to generate the repr. The "name", "loader", "origin", and "has_location" attributes are consulted.
- If the module has a `__file__` attribute, this is used as part of the module's repr.
- If the module has no `__file__` but does have a `__loader__` that is not `None`, then the loader's repr is used as part of the module's repr.
- Otherwise, just use the module's `__name__` in the repr.

Changed in version 3.4: Use of `loader.module_repr()` has been deprecated and the module spec is now used by the import machinery to generate a module repr.

For backward compatibility with Python 3.3, the module repr will be generated by calling the loader's `module_repr()` method, if defined, before trying either approach described above. However, the method is deprecated.

5.4.7. Cached bytecode invalidation

Before Python loads cached bytecode from a `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source's last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source's metadata.

Python also supports "hash-based" cache files, which store a hash of the source file's contents rather than its metadata. There are two variants of hash-based `.pyc` files: checked and unchecked. For checked hash-based `.pyc` files, Python validates the cache file by hashing the source file and comparing the resulting hash with the hash in the cache file. If a checked hash-based cache file is found to be invalid, Python regenerates it and writes a new checked hash-based cache file. For unchecked hash-based `.pyc` files, Python simply assumes the cache file is valid if it exists. Hash-based `.pyc` files validation behavior may be overridden with the `--check-hash-based-pycs` flag.

Changed in version 3.7: Added hash-based `.pyc` files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.5. The Path Based Finder

As mentioned previously, Python comes with several default meta path finders. One of these, called the [path based finder](#) (`PathFinder`), searches an [import path](#), which contains a list of [path entries](#). Each path entry names a location to search for modules.

The path based finder itself doesn't know how to import anything. Instead, it traverses the individual path entries, associating each of them with a path entry finder that knows how to handle that particular kind of path.

The default set of path entry finders implement all the semantics for finding modules on the file system, handling special file types such as Python source code (`.py` files), Python byte code (`.pyc` files) and shared libraries (e.g. `.so` files). When supported by the `zipimport` module in the standard library, the default path entry finders also handle loading all of these file types (other than shared libraries) from zipfiles.

Path entries need not be limited to file system locations. They can refer to URLs, database queries, or any other location that can be specified as a string.

The path based finder provides additional hooks and protocols so that you can extend and customize the types of searchable path entries. For example, if you wanted to support path entries as network URLs, you could write a hook that implements HTTP semantics to find modules on the web. This hook (a callable) would return a [path entry finder](#) supporting the protocol described below, which was then used to get a loader for the module from the web.

A word of warning: this section and the previous both use the term *finder*, distinguishing between them by using the terms [meta path finder](#) and [path entry finder](#). These two types of finders are very similar, support similar protocols, and function in similar ways during the import process, but it's important to keep in mind that they are subtly different. In particular, meta path finders operate at the beginning of the import process, as keyed off of the `sys.meta_path` traversal.

By contrast, path entry finders are in a sense an implementation detail of the path based finder, and in fact, if the path based finder were to be removed from `sys.meta_path`, none of the path entry finder semantics would be invoked.

5.5.1. Path entry finders

The [path based finder](#) is responsible for finding and loading Python modules and packages whose location is specified with a string [path entry](#). Most path entries name locations in the file system, but they need not be limited to this.

As a meta path finder, the [path based finder](#) implements the `find_spec()` protocol previously described, however it exposes additional hooks that can be used to customize how modules are found and loaded from the [import path](#).

Three variables are used by the [path based finder](#), `sys.path`, `sys.path_hooks` and `sys.path_importer_cache`. The `__path__` attributes on package objects are also used. These provide additional ways that the import machinery can be customized.

`sys.path` contains a list of strings providing search locations for modules and packages. It is initialized from the `PYTHONPATH` environment variable and various other installation- and implementation-specific defaults. Entries in `sys.path` can name directories on the file system, zip files, and potentially other “locations” (see the `site` module) that should be searched for modules, such as URLs, or database queries. Only strings and bytes should be present on `sys.path`; all other data types are ignored. The encoding of bytes entries is determined by the individual `path entry finders`.

The `path based finder` is a `meta path finder`, so the import machinery begins the import path search by calling the `path based finder`'s `find_spec()` method as described previously. When the `path` argument to `find_spec()` is given, it will be a list of string paths to traverse - typically a package's `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and `sys.path` is used.

The `path based finder` iterates over every entry in the search path, and for each of these, looks for an appropriate `path entry finder` (`PathEntryFinder`) for the path entry. Because this can be an expensive operation (e.g. there may be `stat()` call overheads for this search), the `path based finder` maintains a cache mapping path entries to `path entry finders`. This cache is maintained in `sys.path_importer_cache` (despite the name, this cache actually stores `finder` objects rather than being limited to `importer` objects). In this way, the expensive search for a particular `path entry` location's `path entry finder` need only be done once. User code is free to remove cache entries from `sys.path_importer_cache` forcing the `path based finder` to perform the `path entry` search again ³.

If the `path entry` is not present in the cache, the `path based finder` iterates over every callable in `sys.path_hooks`. Each of the `path entry hooks` in this list is called with a single argument, the `path entry` to be searched. This callable may either return a `path entry finder` that can handle the `path entry`, or it may raise `ImportError`. An `ImportError` is used by the `path based finder` to signal that the hook cannot find a `path entry finder` for that `path entry`. The exception is ignored and `import path` iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise `ImportError`.

If `sys.path_hooks` iteration ends with no `path entry finder` being returned, then the `path based finder`'s `find_spec()` method will store `None` in `sys.path_importer_cache` (to indicate that there is no finder for this `path entry`) and return `None`, indicating that this `meta path finder` could not find the module.

If a `path entry finder` is returned by one of the `path entry hook` callables on `sys.path_hooks`, then the following protocol is used to ask the `finder` for a `module spec`, which is then used when loading the module.

The current working directory – denoted by an empty string – is handled slightly differently from other entries on `sys.path`. First, if the current working directory is found to not exist, no value is stored in `sys.path_importer_cache`. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for `sys.path_importer_cache` and returned by `importlib.machinery.PathFinder.find_spec()` will be the actual current working directory and not the empty string.

5.5.2. Path entry finder protocol

In order to support imports of modules and initialized packages and also to contribute portions to namespace packages, `path entry finders` must implement the `find_spec()` method.

`find_spec()` takes two arguments: the fully qualified name of the module being imported, and the (optional) target module. `find_spec()` returns a fully populated spec for the module. This spec will always have “loader” set (with one exception).

To indicate to the import machinery that the spec represents a namespace `portion`, the `path entry finder` sets “`submodule_search_locations`” to a list containing the portion.

Changed in version 3.4: `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Older `path entry finders` may implement one of these two deprecated methods instead of `find_spec()`. The methods are still respected for the sake of backward compatibility. However, if `find_spec()` is implemented on the `path entry finder`, the legacy methods are ignored.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace [portion](#).

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional `find_module()` method that meta path finders support. However path entry finder `find_module()` methods are never called with a `path` argument (they are expected to record the appropriate path information from the initial call to the path hook).

The `find_module()` method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both `find_loader()` and `find_module()` exist on a path entry finder, the import system will always call `find_loader()` in preference to `find_module()`.

5.6. Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient. This technique may also be employed at the module level to only alter the behaviour of import statements within that module.

To selectively prevent the import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

5.7. Package Relative Imports

Relative imports use leading dots. A single leading dot indicates a relative import, starting with the current package. Two or more leading dots indicate a relative import to the parent(s) of the current package, one level per dot after the first. For example, given the following package layout:

```
1 package/
2     __init__.py
3     subpackage1/
4         __init__.py
5         moduleX.py
6         moduleY.py
7     subpackage2/
8         __init__.py
9         moduleZ.py
10    moduleA.py
```

In either `subpackage1/moduleX.py` or `subpackage1/__init__.py`, the following are valid relative imports:

```
1 from .moduleY import spam
2 from .moduleY import spam as ham
3 from . import moduleY
4 from ..subpackage1 import moduleY
5 from ..subpackage2.moduleZ import eggs
6 from ..moduleA import foo
```

Absolute imports may use either the `import <>` or `from <> import <>` syntax, but relative imports may only use the second form; the reason for this is that:

```
import XXX.YYY.ZZZ
```

should expose `xxx.yyy.zzz` as a usable expression, but `.moduleY` is not a valid expression.

5.8. Special considerations for `__main__`

The `__main__` module is a special case relative to Python's import system. As noted [elsewhere](#), the `__main__` module is directly initialized at interpreter startup, much like `sys` and `builtins`. However, unlike those two, it doesn't strictly qualify as a built-in module. This is because the manner in which `__main__` is initialized depends on the flags and other options with which the interpreter is invoked.

5.8.1. `__main__.spec`

Depending on how `__main__` is initialized, `__main__.spec` gets set appropriately or to `None`.

When Python is started with the `-m` option, `__spec__` is set to the module spec of the corresponding module or package. `__spec__` is also populated when the `__main__` module is loaded as part of executing a directory, zipfile or other `sys.path` entry.

In [the remaining cases](#) `__main__.spec` is set to `None`, as the code used to populate the `__main__` does not correspond directly with an importable module:

- interactive prompt
- `-c` option
- running from stdin
- running directly from a source or bytecode file

Note that `__main__.spec` is always `None` in the last case, *even if* the file could technically be imported directly as a module instead. Use the `-m` switch if valid module metadata is desired in `__main__`.

Note also that even when `__main__` corresponds with an importable module and `__main__.spec` is set accordingly, they're still considered *distinct* modules. This is due to the fact that blocks guarded by `if __name__ == "__main__":` checks only execute when the module is used to populate the `__main__` namespace, and not during normal import.

Misc

Blog

 ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ys5bq>

Algorithms

 ds-algo (forked) - CodeSandbox

[https://codesandbox.io/s/ds-algo-forked-iezyk?
file=/index.html](https://codesandbox.io/s/ds-algo-forked-iezyk?file=/index.html)

Awesome Find:

Awesome Find:

 ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ds-algo-forked-ysri0>

DS Website

 ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/s2y91>

My Blog



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/c1qdv>

CS-Unit



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/e754i>

<https://codesandbox.io/s/ys5bq>



ds-algo (forked) - CodeSandbox

<https://codesandbox.io/s/ys5bq>

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:

Python's Default Argument Values and Lists

Have you ever written a function that used a list for a default argument value, only to have weird things happen?

```
1 def foo(a=[]):
2     # ... do something with `a` here ...
3     return a
```

And it's not just with lists—the problem manifests with any *mutable* data type when it is used as a default argument value.

Here's what happening, and here's how to fix it.

Check out these two pieces of identical code, one in Python and one in JS.

The code is supposed to append `1` to whatever array you pass in. And return it. And if you don't pass an array, it sets the array to empty by default:

```
1 def foo(a=[]):    # BAD
2     a.append(1)
3     return a
4
5 x = foo()
6 print(x)
7
8 y = foo()
9 print(y)
10
11 z = foo()
12 print(z)
```

and JS:

```
1 function foo(a=[]) {
2     a.push(1);
3     return a;
4 }
5
6 let x = foo();
7 console.log(x);
8
9 let y = foo();
10 console.log(y);
11
12 let z = foo();
13 console.log(z);
```

If I run them, look at the output of the JS, which is as-expected:

```
1 $ node default.js
2 [ 1 ]
3 [ 1 ]
4 [ 1 ]
```

and look at the output of Python, which is not expected!

```
1 $ python default.py
2 [1]
3 [1, 1]
4 [1, 1, 1]
```

What's going on?

This all has to do with *when* the default value is created.

Javascript creates the default empty `[]` *when the function is called*. So each time you call it, it makes a new empty array. Every call returns a different array.

Python creates the default empty `[]` *when the function is loaded*. So it gets created once only when the program is first read into memory, and that's it. There's only one default list no matter how many times you call the function. And so `foo()` is returning *the same list* every time you call it with no arguments. This is why another `1` gets added on each call—`.append(1)` is happening to the same list every time.

Indeed, if you run this in Python:

```
foo() is foo()
```

You'll get `True`, since the same list is being returned.

The fix is to use `None` as a substitute, and then take special action to create a new list on the spot.

```
1 if a is None:
2     a = [] # Make a new list right now, every time the function is called with no args
3     a.append(1)
4     return a
5
6
7 x = foo()
8 print(x)
9
10 y = foo()
11 print(y)
12
13 z = foo()
14 print(z)
15
16 x is y # False, they're different lists, like we wanted
17 y is z # False, they're different lists, like we wanted
```

And then we get good output:

```
1 $ python3 default_good.py
2 [1]
3 [1]
4 [1]
```

Now, if we had a function that used an *immutable* value as a default argument, we have no problem even though the same process is happening.

```
1 def foo(a="hello!"):
2     return a
```

In that code, there's only one `"hello!"`. It gets created when the program is first loaded, and never again. All calls to the function return the same `"hello!"`.

So how is that OK, but it's not OK with a list?

It's because we only ever notice there's a problem when we modify the data. And since we can't modify `"hello!"`, there won't be a problem.

Put another way, we simply don't care if variables are pointing to the same `"hello!"` or to different `"hello!"`s. We cannot tell the difference.

But with something mutable like a list, we certainly can tell, but only after we mutate it and see if it only affects one variable, or if it affects them all.

SCRAP

```
1 """
2 """
3 """
4
5
6 def plus_one(digits):
7     # Your code here
8     new_string = ""
9     for digit in digits:
10         new_digit = str(digit)
11         new_string += new_digit
12     print(new_string)
13     str_tonum = int(new_string)
14     str_tonum = str_tonum + 1
15     str_tonum = str(str_tonum)
16     new_list = list(str_tonum)
17     str_num = []
18     for string in new_list:
19         string = int(string)
20         str_num.append(string)
21     return str_num
22
23
24 print(plus_one([1, 3, 2]))
25 print(plus_one([3, 2, 1, 9]))
26 print(plus_one([9, 9, 9]))
27 # uper
28 # input an array of nums
29 # output an array of all the numbs togheter plus 1
30 # PLAN
31 # add numbs into a string
32 # make string into a number and add 1
33 # put it back into an array and return
34
```

Python Snippets

Calculates the date of `n` days from the given date.

- Use `datetime.timedelta` and the `+` operator to calculate the new `datetime.datetime` value after adding `n` days to `d`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime, timedelta
2
3 def add_days(n, d = datetime.today()):
4     return d + timedelta(n)
```

```
1 from datetime import date
2
3 add_days(5, date(2020, 10, 25)) # date(2020, 10, 30)
4 add_days(-5, date(2020, 10, 25)) # date(2020, 10, 20)
```

Checks if all elements in a list are equal.

- Use `set()` to eliminate duplicate elements and then use `len()` to check if length is `1`.

```
1 def all_equal(lst):
2     return len(set(lst)) == 1
```

```
1 all_equal([1, 2, 3, 4, 5, 6]) # False
2 all_equal([1, 1, 1, 1]) # True
```

Checks if all the values in a list are unique.

- Use `set()` on the given list to keep only unique occurrences.
- Use `len()` to compare the length of the unique values to the original list.

```
1 def all_unique(lst):
2     return len(lst) == len(set(lst))
```

```
1 x = [1, 2, 3, 4, 5, 6]
2 y = [1, 2, 2, 3, 4, 5]
3 all_unique(x) # True
4 all_unique(y) # False
```

Generates a list of numbers in the arithmetic progression starting with the given positive integer and up to the specified limit.

- Use `range()` and `list()` with the appropriate start, step and end values.

```
1 def arithmetic_progression(n, lim):
2     return list(range(n, lim + 1, n))
```

```
arithmetic_progression(5, 25) # [5, 10, 15, 20, 25]
```

Calculates the average of two or more numbers.

- Use `sum()` to sum all of the `args` provided, divide by `len()`.

```
1 def average(*args):
2     return sum(args, 0.0) / len(args)
```

```
1 average(*[1, 2, 3]) # 2.0
2 average(1, 2, 3) # 2.0
```

Calculates the average of a list, after mapping each element to a value using the provided function.

- Use `map()` to map each element to the value returned by `fn`.
- Use `sum()` to sum all of the mapped values, divide by `len(lst)`.
- Omit the last argument, `fn`, to use the default identity function.

```
1 def average_by(lst, fn = lambda x: x):
2     return sum(map(fn, lst), 0.0) / len(lst)
```

```
1 average_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda x: x['n'])
2 # 5.0
```

Splits values into two groups, based on the result of the given `filter` list.

- Use a list comprehension and `zip()` to add elements to groups, based on `filter`.
- If `filter` has a truthy value for any element, add it to the first group, otherwise add it to the second group.

```
1 def bifurcate(lst, filter):
2     return [
3         [x for x, flag in zip(lst, filter) if flag],
4         [x for x, flag in zip(lst, filter) if not flag]
5     ]
```

```
1 bifurcate(['beep', 'boop', 'foo', 'bar'], [True, True, False, True])
2 # [[ 'beep', 'boop', 'bar' ], [ 'foo' ] ]
```

Splits values into two groups, based on the result of the given filtering function.

- Use a list comprehension to add elements to groups, based on the value returned by `fn` for each element.

- If `fn` returns a truthy value for any element, add it to the first group, otherwise add it to the second group.

```

1 def bifurcate_by(lst, fn):
2     return [
3         [x for x in lst if fn(x)],
4         [x for x in lst if not fn(x)]
5     ]

```

```

1 bifurcate_by(['beep', 'boop', 'foo', 'bar'], lambda x: x[0] == 'b')
2 # [[ 'beep', 'boop', 'bar'], ['foo'] ]

```

Calculates the number of ways to choose `k` items from `n` items without repetition and without order.

- Use `math.comb()` to calculate the binomial coefficient.

```

1 from math import comb
2
3 def binomial_coefficient(n, k):
4     return comb(n, k)

```

```
binomial_coefficient(8, 2) # 28
```

Returns the length of a string in bytes.

- Use `str.encode('utf-8')` to encode the given string and return its length.

```

1 def byte_size(s):
2     return len(s.encode('utf-8'))

```

```

1 byte_size(' ') # 4
2 byte_size('Hello World') # 11

```

Converts a string to camelcase.

- Use `re.sub()` to replace any `-` or `_` with a space, using the regexp `r"(_|-)+"`.
- Use `str.title()` to capitalize the first letter of each word and convert the rest to lowercase.
- Finally, use `str.replace()` to remove spaces between words.

```

1 from re import sub
2
3 def camel(s):
4     s = sub(r"(_|-)+", " ", s).title().replace(" ", "")
5     return ''.join([s[0].lower(), s[1:]])

```

```
1 camel('some_database_field_name') # 'someDatabaseFieldName'
```

```
2 camel('Some label that needs to be camelized')
3 # 'someLabelThatNeedsToBeCamelized'
4 camel('some-javascript-property') # 'someJavascriptProperty'
5 camel('some-mixed_string with spaces_underscores-and-hyphens')
6 # 'someMixedStringWithSpacesUnderscoresAndHyphens'
```

Capitalizes the first letter of a string.

- Use list slicing and `str.upper()` to capitalize the first letter of the string.
- Use `str.join()` to combine the capitalized first letter with the rest of the characters.
- Omit the `lower_rest` parameter to keep the rest of the string intact, or set it to `True` to convert to lowercase.

```
1 def capitalize(s, lower_rest = False):
2     return ''.join([s[:1].upper(), (s[1:]).lower() if lower_rest else s[1:]])
```

```
1 capitalize('fooBar') # 'FooBar'
2 capitalize('fooBar', True) # 'Foobar'
```

Capitalizes the first letter of every word in a string.

- Use `str.title()` to capitalize the first letter of every word in the string.

```
1 def capitalize_every_word(s):
2     return s.title()
```

```
capitalize_every_word('hello world!') # 'Hello World!'
```

Casts the provided value as a list if it's not one.

- Use `isinstance()` to check if the given value is enumerable.
- Return it by using `list()` or encapsulated in a list accordingly.

```
1 def cast_list(val):
2     return list(val) if isinstance(val, (tuple, list, set, dict)) else [val]
```

```
1 cast_list('foo') # ['foo']
2 cast_list([1]) # [1]
3 cast_list(('foo', 'bar')) # ['foo', 'bar']
```

unlisted: true

Converts Celsius to Fahrenheit.

- Follow the conversion formula $F = 1.8 * C + 32$.

```
1 def celsius_to_fahrenheit(degrees):
2     return ((degrees * 1.8) + 32)
```

```
celsius_to_fahrenheit(180) # 356.0
```

Creates a function that will invoke a predicate function for the specified property on a given object.

- Return a `lambda` function that takes an object and applies the predicate function, `fn` to the specified property.

```
1 def check_prop(fn, prop):
2     return lambda obj: fn(obj[prop])
```

```
1 check_age = check_prop(lambda x: x >= 18, 'age')
2 user = {'name': 'Mark', 'age': 18}
3 check_age(user) # True
```

Chunks a list into smaller lists of a specified size.

- Use `list()` and `range()` to create a list of the desired `size`.
- Use `map()` on the list and fill it with splices of the given list.
- Finally, return the created list.

```
1 from math import ceil
2
3 def chunk(lst, size):
4     return list(
5         map(lambda x: lst[x * size:x * size + size],
6             list(range(ceil(len(lst) / size)))))
```

```
chunk([1, 2, 3, 4, 5], 2) # [[1, 2], [3, 4], [5]]
```

Chunks a list into `n` smaller lists.

- Use `math.ceil()` and `len()` to get the size of each chunk.
- Use `list()` and `range()` to create a new list of size `n`.
- Use `map()` to map each element of the new list to a chunk the length of `size`.
- If the original list can't be split evenly, the final chunk will contain the remaining elements.

```
1 from math import ceil
2
3 def chunk_into_n(lst, n):
4     size = ceil(len(lst) / n)
5     return list(
6         map(lambda x: lst[x * size:x * size + size],
7             list(range(n)))
8     )
```

```
chunk_into_n([1, 2, 3, 4, 5, 6, 7], 4) # [[1, 2], [3, 4], [5, 6], [7]]
```

Clamps `num` within the inclusive range specified by the boundary values.

- If `num` falls within the range (`a`, `b`), return `num`.
- Otherwise, return the nearest number in the range.

```
1 def clamp_number(num, a, b):  
2     return max(min(num, max(a, b)), min(a, b))
```

```
1 clamp_number(2, 3, 5) # 3  
2 clamp_number(1, -1, -5) # -1
```

Inverts a dictionary with non-unique hashable values.

- Create a `collections.defaultdict` with `list` as the default value for each key.
- Use `dictionary.items()` in combination with a loop to map the values of the dictionary to keys using `dict.append()`.
- Use `dict()` to convert the `collections.defaultdict` to a regular dictionary.

```
1 from collections import defaultdict  
2  
3 def collect_dictionary(obj):  
4     inv_obj = defaultdict(list)  
5     for key, value in obj.items():  
6         inv_obj[value].append(key)  
7     return dict(inv_obj)
```

```
1 ages = {  
2     'Peter': 10,  
3     'Isabel': 10,  
4     'Anna': 9,  
5 }  
6 collect_dictionary(ages) # { 10: ['Peter', 'Isabel'], 9: ['Anna'] }
```

Combines two or more dictionaries, creating a list of values for each key.

- Create a new `collections.defaultdict` with `list` as the default value for each key and loop over `dicts`.
- Use `dict.append()` to map the values of the dictionary to keys.
- Use `dict()` to convert the `collections.defaultdict` to a regular dictionary.

```
1 from collections import defaultdict  
2  
3 def combine_values(*dicts):  
4     res = defaultdict(list)  
5     for d in dicts:  
6         for key in d:  
7             res[key].append(d[key])  
8     return dict(res)
```

```
1 d1 = {'a': 1, 'b': 'foo', 'c': 400}
2 d2 = {'a': 3, 'b': 200, 'd': 400}
3
4 combine_values(d1, d2) # {'a': [1, 3], 'b': ['foo', 200], 'c': [400], 'd': [400]}
```

Removes falsy values from a list.

- Use `filter()` to filter out falsy values (`False`, `None`, `0`, and `""`).

```
1 def compact(lst):
2     return list(filter(None, lst))
```

```
compact([0, 1, False, 2, '', 3, 'a', 's', 34]) # [ 1, 2, 3, 'a', 's', 34 ]
```

Performs right-to-left function composition.

- Use `functools.reduce()` to perform right-to-left function composition.
- The last (rightmost) function can accept one or more arguments; the remaining functions must be unary.

```
1 from functools import reduce
2
3 def compose(*fns):
4     return reduce(lambda f, g: lambda *args: f(g(*args)), fns)
```

```
1 add5 = lambda x: x + 5
2 multiply = lambda x, y: x * y
3 multiply_and_add_5 = compose(add5, multiply)
4 multiply_and_add_5(5, 2) # 15
```

Performs left-to-right function composition.

- Use `functools.reduce()` to perform left-to-right function composition.
- The first (leftmost) function can accept one or more arguments; the remaining functions must be unary.

```
1 from functools import reduce
2
3 def compose_right(*fns):
4     return reduce(lambda f, g: lambda *args: g(f(*args)), fns)
```

```
1 add = lambda x, y: x + y
2 square = lambda x: x * x
3 add_and_square = compose_right(add, square)
4 add_and_square(1, 2) # 9
```

Groups the elements of a list based on the given function and returns the count of elements in each group.

- Use `collections.defaultdict` to initialize a dictionary.
- Use `map()` to map the values of the given list using the given function.
- Iterate over the map and increase the element count each time it occurs.

```

1 from collections import defaultdict
2
3 def count_by(lst, fn = lambda x: x):
4     count = defaultdict(int)
5     for val in map(fn, lst):
6         count[val] += 1
7     return dict(count)

```

```

1 from math import floor
2
3 count_by([6.1, 4.2, 6.3], floor) # {6: 2, 4: 1}
4 count_by(['one', 'two', 'three'], len) # {3: 2, 5: 1}

```

Counts the occurrences of a value in a list.

- Use `list.count()` to count the number of occurrences of `val` in `lst`.

```

1 def count_occurrences(lst, val):
2     return lst.count(val)

```

```
count_occurrences([1, 1, 2, 1, 2, 3], 1) # 3
```

Creates a list of partial sums.

- Use `itertools.accumulate()` to create the accumulated sum for each element.
- Use `list()` to convert the result into a list.

```

1 from itertools import accumulate
2
3 def cumsum(lst):
4     return list(accumulate(lst))

```

```
cumsum(range(0, 15, 3)) # [0, 3, 9, 18, 30]
```

Curries a function.

- Use `functools.partial()` to return a new partial object which behaves like `fn` with the given arguments, `args`, partially applied.

```

1 from functools import partial
2
3 def curry(fn, *args):
4     return partial(fn, *args)

```

```
1 add = lambda x, y: x + y
2 add10 = curry(add, 10)
3 add10(20) # 30
```

Creates a list of dates between `start` (inclusive) and `end` (not inclusive).

- Use `datetime.timedelta.days` to get the days between `start` and `end`.
- Use `int()` to convert the result to an integer and `range()` to iterate over each day.
- Use a list comprehension and `datetime.timedelta()` to create a list of `datetime.date` objects.

```
1 from datetime import timedelta, date
2
3 def daterange(start, end):
4     return [start + timedelta(n) for n in range(int((end - start).days))]
```

```
1 from datetime import date
2
3 daterange(date(2020, 10, 1), date(2020, 10, 5))
4 # [date(2020, 10, 1), date(2020, 10, 2), date(2020, 10, 3), date(2020, 10, 4)]
```

Calculates the date of `n` days ago from today.

- Use `datetime.date.today()` to get the current day.
- Use `datetime.timedelta` to subtract `n` days from today's date.

```
1 from datetime import timedelta, date
2
3 def days_ago(n):
4     return date.today() - timedelta(n)
```

```
days_ago(5) # date(2020, 10, 23)
```

Calculates the day difference between two dates.

- Subtract `start` from `end` and use `datetime.timedelta.days` to get the day difference.

```
1 def days_diff(start, end):
2     return (end - start).days
```

```
1 from datetime import date
2
3 days_diff(date(2020, 10, 25), date(2020, 10, 28)) # 3
```

Calculates the date of `n` days from today.

- Use `datetime.date.today()` to get the current day.
- Use `datetime.timedelta` to add `n` days from today's date.

```
1 from datetime import timedelta, date
2
3 def days_from_now(n):
4     return date.today() + timedelta(n)
```

```
days_from_now(5) # date(2020, 11, 02)
```

Decapitalizes the first letter of a string.

- Use list slicing and `str.lower()` to decapitalize the first letter of the string.
- Use `str.join()` to combine the lowercase first letter with the rest of the characters.
- Omit the `upper_rest` parameter to keep the rest of the string intact, or set it to `True` to convert to uppercase.

```
1 def decapitalize(s, upper_rest = False):
2     return ''.join([s[:1].lower(), (s[1:]).upper() if upper_rest else s[1:]])
```

```
1 decapitalize('FooBar') # 'fooBar'
2 decapitalize('FooBar', True) # 'FOOBAR'
```

Deep flattens a list.

- Use recursion.
- Use `isinstance()` with `collections.abc.Iterable` to check if an element is iterable.
- If it is iterable, apply `deep_flatten()` recursively, otherwise return `[lst]`.

```
1 from collections.abc import Iterable
2
3 def deep_flatten(lst):
4     return ([a for i in lst for a in
5             deep_flatten(i)] if isinstance(lst, Iterable) else [lst])
```

```
deep_flatten([1, [2], [[3], 4], 5]) # [1, 2, 3, 4, 5]
```

Converts an angle from degrees to radians.

- Use `math.pi` and the degrees to radians formula to convert the angle from degrees to radians.

```
1 from math import pi
2
3 def degrees_to_rads(deg):
4     return (deg * pi) / 180.0
```

```
degrees_to_rads(180) # ~3.1416
```

Invokes the provided function after `ms` milliseconds.

- Use `time.sleep()` to delay the execution of `fn` by `ms / 1000` seconds.

```
1 from time import sleep
2
3 def delay(fn, ms, *args):
4     sleep(ms / 1000)
5     return fn(*args)
```

```
delay(lambda x: print(x), 1000, 'later') # prints 'later' after one second
```

Converts a dictionary to a list of tuples.

- Use `dict.items()` and `list()` to get a list of tuples from the given dictionary.

```
1 def dict_to_list(d):
2     return list(d.items())
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 dict_to_list(d)
3 # [('one', 1), ('three', 3), ('five', 5), ('two', 2), ('four', 4)]
```

Calculates the difference between two iterables, without filtering duplicate values.

- Create a `set` from `b`.
- Use a list comprehension on `a` to only keep values not contained in the previously created set, `_b`.

```
1 def difference(a, b):
2     _b = set(b)
3     return [item for item in a if item not in _b]
```

```
difference([1, 2, 3], [1, 2, 4]) # [3]
```

Returns the difference between two lists, after applying the provided function to each list element of both.

- Create a `set`, using `map()` to apply `fn` to each element in `b`.
- Use a list comprehension in combination with `fn` on `a` to only keep values not contained in the previously created set, `_b`.

```
1 def difference_by(a, b, fn):
2     _b = set(map(fn, b))
3     return [item for item in a if fn(item) not in _b]
```

```
1 from math import floor
2
3 difference_by([2.1, 1.2], [2.3, 3.4], floor) # [1.2]
4 difference_by([{ 'x': 2 }, { 'x': 1 }], [{ 'x': 1 }], lambda v : v['x'])
5 # [ { x: 2 } ]
```

Converts a number to a list of digits.

- Use `map()` combined with `int` on the string representation of `n` and return a list from the result.

```
1 def digitize(n):
2     return list(map(int, str(n)))
```

```
digitize(123) # [1, 2, 3]
```

Returns a list with `n` elements removed from the left.

- Use slice notation to remove the specified number of elements from the left.
- Omit the last argument, `n`, to use a default value of `1`.

```
1 def drop(a, n = 1):
2     return a[n:]
```

```
1 drop([1, 2, 3]) # [2, 3]
2 drop([1, 2, 3], 2) # [3]
3 drop([1, 2, 3], 42) # []
```

Returns a list with `n` elements removed from the right.

- Use slice notation to remove the specified number of elements from the right.
- Omit the last argument, `n`, to use a default value of `1`.

```
1 def drop_right(a, n = 1):
2     return a[:-n]
```

```
1 drop_right([1, 2, 3]) # [1, 2]
2 drop_right([1, 2, 3], 2) # [1]
3 drop_right([1, 2, 3], 42) # []
```

Checks if the provided function returns `True` for every element in the list.

- Use `all()` in combination with `map()` and `fn` to check if `fn` returns `True` for all elements in the list.

```
1 def every(lst, fn = lambda x: x):
```

```
2     return all(map(fn, lst))
```

```
1 every([4, 2, 3], lambda x: x > 1) # True
2 every([1, 2, 3]) # True
```

Returns every `nth` element in a list.

- Use slice notation to create a new list that contains every `nth` element of the given list.

```
1 def every_nth(lst, nth):
2     return lst[nth - 1::nth]
```

```
every_nth([1, 2, 3, 4, 5, 6], 2) # [ 2, 4, 6 ]
```

Calculates the factorial of a number.

- Use recursion.
- If `num` is less than or equal to `1`, return `1`.
- Otherwise, return the product of `num` and the factorial of `num - 1`.
- Throws an exception if `num` is a negative or a floating point number.

```
1 def factorial(num):
2     if not ((num >= 0) and (num % 1 == 0)):
3         raise Exception("Number can't be floating point or negative.")
4     return 1 if num == 0 else num * factorial(num - 1)
```

```
factorial(6) # 720
```

unlisted: true

Converts Fahrenheit to Celsius.

- Follow the conversion formula $C = (F - 32) \times 5/9$.

```
1 def fahrenheit_to_celsius(degrees):
2     return ((degrees - 32) * 5/9)
```

```
fahrenheit_to_celsius(77) # 25.0
```

Generates a list, containing the Fibonacci sequence, up until the `nth` term.

- Starting with `0` and `1`, use `list.append()` to add the sum of the last two numbers of the list to the end of the list, until the length of the list reaches `n`.

- If `n` is less or equal to `0`, return a list containing `0`.

```

1 def fibonacci(n):
2     if n <= 0:
3         return [0]
4     sequence = [0, 1]
5     while len(sequence) <= n:
6         next_value = sequence[len(sequence) - 1] + sequence[len(sequence) - 2]
7         sequence.append(next_value)
8     return sequence

```

```
fibonacci(7) # [0, 1, 1, 2, 3, 5, 8, 13]
```

Creates a list with the non-unique values filtered out.

- Use `collections.Counter` to get the count of each value in the list.
- Use a list comprehension to create a list containing only the unique values.

```

1 from collections import Counter
2
3 def filter_non_unique(lst):
4     return [item for item, count in Counter(lst).items() if count == 1]

```

```
filter_non_unique([1, 2, 2, 3, 4, 4, 5]) # [1, 3, 5]
```

Creates a list with the unique values filtered out.

- Use `collections.Counter` to get the count of each value in the list.
- Use a list comprehension to create a list containing only the non-unique values.

```

1 from collections import Counter
2
3 def filter_unique(lst):
4     return [item for item, count in Counter(lst).items() if count > 1]

```

```
filter_unique([1, 2, 2, 3, 4, 4, 5]) # [2, 4]
```

Finds the value of the first element in the given list that satisfies the provided testing function.

- Use a list comprehension and `next()` to return the first element in `lst` for which `fn` returns `True`.

```

1 def find(lst, fn):
2     return next(x for x in lst if fn(x))

```

```
find([1, 2, 3, 4], lambda n: n % 2 == 1) # 1
```

Finds the index of the first element in the given list that satisfies the provided testing function.

- Use a list comprehension, `enumerate()` and `next()` to return the index of the first element in `lst` for which `fn` returns `True`.

```
1 def find_index(lst, fn):
2     return next(i for i, x in enumerate(lst) if fn(x))
```

```
find_index([1, 2, 3, 4], lambda n: n % 2 == 1) # 0
```

Finds the indexes of all elements in the given list that satisfy the provided testing function.

- Use `enumerate()` and a list comprehension to return the indexes of the all element in `lst` for which `fn` returns `True`.

```
1 def find_index_of_all(lst, fn):
2     return [i for i, x in enumerate(lst) if fn(x)]
```

```
find_index_of_all([1, 2, 3, 4], lambda n: n % 2 == 1) # [0, 2]
```

Finds the first key in the provided dictionary that has the given value.

- Use `dictionary.items()` and `next()` to return the first key that has a value equal to `val`.

```
1 def find_key(dict, val):
2     return next(key for key, value in dict.items() if value == val)
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 find_key(ages, 11) # 'Isabel'
```

Finds all keys in the provided dictionary that have the given value.

- Use `dictionary.items()`, a generator and `list()` to return all keys that have a value equal to `val`.

```
1 def find_keys(dict, val):
2     return list(key for key, value in dict.items() if value == val)
```

```
1 ages = {
2     'Peter': 10,
```

```
3     'Isabel': 11,
4     'Anna': 10,
5   }
6 find_keys(ages, 10) # [ 'Peter', 'Anna' ]
```

Finds the value of the last element in the given list that satisfies the provided testing function.

- Use a list comprehension and `next()` to return the last element in `lst` for which `fn` returns `True`.

```
1 def find_last(lst, fn):
2     return next(x for x in lst[::-1] if fn(x))
```

```
find_last([1, 2, 3, 4], lambda n: n % 2 == 1) # 3
```

Finds the index of the last element in the given list that satisfies the provided testing function.

- Use a list comprehension, `enumerate()` and `next()` to return the index of the last element in `lst` for which `fn` returns `True`.

```
1 def find_last_index(lst, fn):
2     return len(lst) - 1 - next(i for i, x in enumerate(lst[::-1]) if fn(x))
```

```
find_last_index([1, 2, 3, 4], lambda n: n % 2 == 1) # 2
```

Finds the items that are parity outliers in a given list.

- Use `collections.Counter` with a list comprehension to count even and odd values in the list.
- Use `collections.Counter.most_common()` to get the most common parity.
- Use a list comprehension to find all elements that do not match the most common parity.

```
1 from collections import Counter
2
3 def find_parity_outliers(nums):
4     return [
5         x for x in nums
6         if x % 2 != Counter([n % 2 for n in nums]).most_common()[0][0]
7     ]
```

```
find_parity_outliers([1, 2, 3, 4, 6]) # [1, 3]
```

Flattens a list of lists once.

- Use a list comprehension to extract each value from sub-lists in order.

```
1 def flatten(lst):
2     return [x for y in lst for x in y]
```

```
flatten([[1, 2, 3, 4], [5, 6, 7, 8]]) # [1, 2, 3, 4, 5, 6, 7, 8]
```

Executes the provided function once for each list element.

- Use a `for` loop to execute `fn` for each element in `itr`.

```
1 def for_each(itr, fn):
2     for el in itr:
3         fn(el)
```

```
for_each([1, 2, 3], print) # 1 2 3
```

Executes the provided function once for each list element, starting from the list's last element.

- Use a `for` loop in combination with slice notation to execute `fn` for each element in `itr`, starting from the last one.

```
1 def for_each_right(itr, fn):
2     for el in itr[::-1]:
3         fn(el)
```

```
for_each_right([1, 2, 3], print) # 3 2 1
```

Creates a dictionary with the unique values of a list as keys and their frequencies as the values.

- Use `collections.defaultdict()` to store the frequencies of each unique element.
- Use `dict()` to return a dictionary with the unique elements of the list as keys and their frequencies as the values.

```
1 from collections import defaultdict
2
3 def frequencies(lst):
4     freq = defaultdict(int)
5     for val in lst:
6         freq[val] += 1
7     return dict(freq)
```

```
frequencies(['a', 'b', 'a', 'c', 'a', 'a', 'b']) # { 'a': 4, 'b': 2, 'c': 1 }
```

Converts a date from its ISO-8601 representation.

- Use `datetime.datetime.fromisoformat()` to convert the given ISO-8601 date to a `datetime.datetime` object.

```
1 from datetime import datetime
2
3 def from_iso_date(d):
4     return datetime.fromisoformat(d)
```

```
from_iso_date('2020-10-28T12:30:59.000000') # 2020-10-28 12:30:59
```

Calculates the greatest common divisor of a list of numbers.

- Use `functools.reduce()` and `math.gcd()` over the given list.

```
1 from functools import reduce
2 from math import gcd as _gcd
3
4 def gcd(numbers):
5     return reduce(_gcd, numbers)
```

```
gcd([8, 36, 28]) # 4
```

Initializes a list containing the numbers in the specified range where `start` and `end` are inclusive and the ratio between two terms is `step`.

Returns an error if `step` equals `1`.

- Use `range()`, `math.log()` and `math.floor()` and a list comprehension to create a list of the appropriate length, applying the step for each element.
- Omit the second argument, `start`, to use a default value of `1`.
- Omit the third argument, `step`, to use a default value of `2`.

```
1 from math import floor, log
2
3 def geometric_progression(end, start=1, step=2):
4     return [start * step ** i for i in range(floor(log(end / start)
5         / log(step)) + 1)]
```

```
1 geometric_progression(256) # [1, 2, 4, 8, 16, 32, 64, 128, 256]
2 geometric_progression(256, 3) # [3, 6, 12, 24, 48, 96, 192]
3 geometric_progression(256, 1, 4) # [1, 4, 16, 64, 256]
```

Retrieves the value of the nested key indicated by the given selector list from a dictionary or list.

- Use `functools.reduce()` to iterate over the `selectors` list.
- Apply `operatorgetitem()` for each key in `selectors`, retrieving the value to be used as the iteratee for the next iteration.

```
1 from functools import reduce
2 from operator import getitem
3
4 def get(d, selectors):
5     return reduce(getitem, selectors, d)
```

```

1 users = {
2     'freddy': {
3         'name': {
4             'first': 'fred',
5             'last': 'smith'
6         },
7         'postIds': [1, 2, 3]
8     }
9 }
10 get(users, ['freddy', 'name', 'last']) # 'smith'
11 get(users, ['freddy', 'postIds', 1]) # 2

```

Groups the elements of a list based on the given function.

- Use `collections.defaultdict` to initialize a dictionary.
- Use `fn` in combination with a `for` loop and `dict.append()` to populate the dictionary.
- Use `dict()` to convert it to a regular dictionary.

```

1 from collections import defaultdict
2
3 def group_by(lst, fn):
4     d = defaultdict(list)
5     for el in lst:
6         d[fn(el)].append(el)
7     return dict(d)

```

```

1 from math import floor
2
3 group_by([6.1, 4.2, 6.3], floor) # {4: [4.2], 6: [6.1, 6.3]}
4 group_by(['one', 'two', 'three'], len) # {3: ['one', 'two'], 5: ['three']}

```

Calculates the Hamming distance between two values.

- Use the XOR operator (`^`) to find the bit difference between the two numbers.
- Use `bin()` to convert the result to a binary string.
- Convert the string to a list and use `count()` of `str` class to count and return the number of `1`s in it.

```

1 def hamming_distance(a, b):
2     return bin(a ^ b).count('1')

```

```
hamming_distance(2, 3) # 1
```

Checks if there are duplicate values in a flat list.

- Use `set()` on the given list to remove duplicates, compare its length with the length of the list.

```

1 def has_duplicates(lst):
2     return len(lst) != len(set(lst))

```

```
1 x = [1, 2, 3, 4, 5, 5]
2 y = [1, 2, 3, 4, 5]
3 has_duplicates(x) # True
4 has_duplicates(y) # False
```

Checks if two lists contain the same elements regardless of order.

- Use `set()` on the combination of both lists to find the unique values.
- Iterate over them with a `for` loop comparing the `count()` of each unique value in each list.
- Return `False` if the counts do not match for any element, `True` otherwise.

```
1 def have_same_contents(a, b):
2     for v in set(a + b):
3         if a.count(v) != b.count(v):
4             return False
5     return True
```

```
have_same_contents([1, 2, 4], [2, 4, 1]) # True
```

Returns the head of a list.

- Use `lst[0]` to return the first element of the passed list.

```
1 def head(lst):
2     return lst[0]
```

```
head([1, 2, 3]) # 1
```

Converts a hexadecimal color code to a tuple of integers corresponding to its RGB components.

- Use a list comprehension in combination with `int()` and list slice notation to get the RGB components from the hexadecimal string.
- Use `tuple()` to convert the resulting list to a tuple.

```
1 def hex_to_rgb(hex):
2     return tuple(int(hex[i:i+2], 16) for i in (0, 2, 4))
```

```
hex_to_rgb('FFA501') # (255, 165, 1)
```

Checks if the given number falls within the given range.

- Use arithmetic comparison to check if the given number is in the specified range.
- If the second parameter, `end`, is not specified, the range is considered to be from `0` to `start`.

```
1 def in_range(n, start, end = 0):
2     return start <= n <= end if end >= start else end <= n <= start
```

```
1 in_range(3, 2, 5) # True
2 in_range(3, 4) # True
3 in_range(2, 3, 5) # False
4 in_range(3, 2) # False
```

Checks if all the elements in `values` are included in `lst`.

- Check if every value in `values` is contained in `lst` using a `for` loop.
- Return `False` if any one value is not found, `True` otherwise.

```
1 def includes_all(lst, values):
2     for v in values:
3         if v not in lst:
4             return False
5     return True
```

```
1 includes_all([1, 2, 3, 4], [1, 4]) # True
2 includes_all([1, 2, 3, 4], [1, 5]) # False
```

Checks if any element in `values` is included in `lst`.

- Check if any value in `values` is contained in `lst` using a `for` loop.
- Return `True` if any one value is found, `False` otherwise.

```
1 def includes_any(lst, values):
2     for v in values:
3         if v in lst:
4             return True
5     return False
```

```
1 includes_any([1, 2, 3, 4], [2, 9]) # True
2 includes_any([1, 2, 3, 4], [8, 9]) # False
```

Returns a list of indexes of all the occurrences of an element in a list.

- Use `enumerate()` and a list comprehension to check each element for equality with `value` and adding `i` to the result.

```
1 def index_of_all(lst, value):
2     return [i for i, x in enumerate(lst) if x == value]
```

```
1 index_of_all([1, 2, 1, 4, 5, 1], 1) # [0, 2, 5]
2 index_of_all([1, 2, 3, 4], 6) # []
```

Returns all the elements of a list except the last one.

- Use `lst[:-1]` to return all but the last element of the list.

```
1 def initial(lst):
2     return lst[:-1]
```

```
initial([1, 2, 3]) # [1, 2]
```

Initializes a 2D list of given width and height and value.

- Use a list comprehension and `range()` to generate `h` rows where each is a list with length `h`, initialized with `val`.
- Omit the last argument, `val`, to set the default value to `None`.

```
1 def initialize_2d_list(w, h, val = None):
2     return [[val for x in range(w)] for y in range(h)]
```

```
initialize_2d_list(2, 2, 0) # [[0, 0], [0, 0]]
```

Initializes a list containing the numbers in the specified range where `start` and `end` are inclusive with their common difference `step`.

- Use `list()` and `range()` to generate a list of the appropriate length, filled with the desired values in the given range.
- Omit `start` to use the default value of `0`.
- Omit `step` to use the default value of `1`.

```
1 def initialize_list_with_range(end, start = 0, step = 1):
2     return list(range(start, end + 1, step))
```

```
1 initialize_list_with_range(5) # [0, 1, 2, 3, 4, 5]
2 initialize_list_with_range(7, 3) # [3, 4, 5, 6, 7]
3 initialize_list_with_range(9, 0, 2) # [0, 2, 4, 6, 8]
```

Initializes and fills a list with the specified value.

- Use a list comprehension and `range()` to generate a list of length equal to `n`, filled with the desired values.
- Omit `val` to use the default value of `0`.

```
1 def initialize_list_with_values(n, val = 0):
2     return [val for x in range(n)]
```

```
initialize_list_with_values(5, 2) # [2, 2, 2, 2, 2]
```

Returns a list of elements that exist in both lists.

- Create a `set` from `a` and `b`.
- Use the built-in set operator `&` to only keep values contained in both sets, then transform the `set` back into a `list`.

```
1 def intersection(a, b):
2     _a, _b = set(a), set(b)
3     return list(_a & _b)
```

```
intersection([1, 2, 3], [4, 3, 2]) # [2, 3]
```

Returns a list of elements that exist in both lists, after applying the provided function to each list element of both.

- Create a `set`, using `map()` to apply `fn` to each element in `b`.
- Use a list comprehension in combination with `fn` on `a` to only keep values contained in both lists.

```
1 def intersection_by(a, b, fn):
2     _b = set(map(fn, b))
3     return [item for item in a if fn(item) in _b]
```

```
1 from math import floor
2
3 intersection_by([2.1, 1.2], [2.3, 3.4], floor) # [2.1]
```

Inverts a dictionary with unique hashable values.

- Use `dictionary.items()` in combination with a list comprehension to create a new dictionary with the values and keys inverted.

```
1 def invert_dictionary(obj):
2     return { value: key for key, value in obj.items() }
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 invert_dictionary(ages) # { 10: 'Peter', 11: 'Isabel', 9: 'Anna' }
```

Checks if a string is an anagram of another string (case-insensitive, ignores spaces, punctuation and special characters).

- Use `str.isalnum()` to filter out non-alphanumeric characters, `str.lower()` to transform each character to lowercase.
- Use `collections.Counter` to count the resulting characters for each string and compare the results.

```
1 from collections import Counter
2
3 def is_anagram(s1, s2):
4     return Counter(
5         c.lower() for c in s1 if c.isalnum()
```

```
6     ) == Counter(
7         c.lower() for c in s2 if c.isalnum()
8     )
```

```
is_anagram('#anagram', 'Nag a ram!') # True
```

Checks if the elements of the first list are contained in the second one regardless of order.

- Use `count()` to check if any value in `a` has more occurrences than it has in `b`.
- Return `False` if any such value is found, `True` otherwise.

```
1 def isContainedIn(a, b):
2     for v in set(a):
3         if a.count(v) > b.count(v):
4             return False
5     return True
```

```
isContainedIn([1, 4], [2, 4, 1]) # True
```

unlisted: true

Checks if the first numeric argument is divisible by the second one.

- Use the modulo operator (`%`) to check if the remainder is equal to `0`.

```
1 def isDivisible(dividend, divisor):
2     return dividend % divisor == 0
```

```
isDivisible(6, 3) # True
```

unlisted: true

Checks if the given number is even.

- Check whether a number is odd or even using the modulo (`%`) operator.
- Return `True` if the number is even, `False` if the number is odd.

```
1 def isEven(num):
2     return num % 2 == 0
```

```
isEven(3) # False
```

unlisted: true

Checks if the given number is odd.

- Checks whether a number is even or odd using the modulo (`%`) operator.
- Returns `True` if the number is odd, `False` if the number is even.

```
1 def is_odd(num):  
2     return num % 2 != 0
```

```
is_odd(3) # True
```

Checks if the provided integer is a prime number.

- Return `False` if the number is `0`, `1`, a negative number or a multiple of `2`.
- Use `all()` and `range()` to check numbers from `3` to the square root of the given number.
- Return `True` if none divides the given number, `False` otherwise.

```
1 from math import sqrt  
2  
3 def is_prime(n):  
4     if n <= 1 or (n % 2 == 0 and n > 2):  
5         return False  
6     return all(n % i for i in range(3, int(sqrt(n)) + 1, 2))
```

```
is_prime(11) # True
```

Checks if the given date is a weekday.

- Use `datetime.datetime.weekday()` to get the day of the week as an integer.
- Check if the day of the week is less than or equal to `4`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime  
2  
3 def is_weekday(d = datetime.today()):  
4     return d.weekday() <= 4
```

```
1 from datetime import date  
2  
3 is_weekday(date(2020, 10, 25)) # False  
4 is_weekday(date(2020, 10, 28)) # True
```

Checks if the given date is a weekend.

- Use `datetime.datetime.weekday()` to get the day of the week as an integer.
- Check if the day of the week is greater than `4`.
- Omit the second argument, `d`, to use a default value of `datetime.today()`.

```
1 from datetime import datetime
2
3 def is_weekend(d = datetime.today()):
4     return d.weekday() > 4
```

```
1 from datetime import date
2
3 is_weekend(date(2020, 10, 25)) # True
4 is_weekend(date(2020, 10, 28)) # False
```

Converts a string to kebab case.

- Use `re.sub()` to replace any `-` or `_` with a space, using the regexp `r"(_|-)+"`.
- Use `re.sub()` to match all words in the string, `str.lower()` to lowercase them.
- Finally, use `str.join()` to combine all word using `-` as the separator.

```
1 from re import sub
2
3 def kebab(s):
4     return '-'.join(
5         sub(r"(\s_|-)+", " ",
6             sub(r"[A-Z]{2,}(?=[A-Z][a-z]+[0-9]*|\b)|[A-Z]?[a-z]+[0-9]*|[A-Z]|[0-9]+",
7                 lambda mo: ' ' + mo.group(0).lower(), s)).split())
```

```
1 kebab('camelCase') # 'camel-case'
2 kebab('some text') # 'some-text'
3 kebab('some-mixed_string With spaces_underscores-and-hyphens')
4 # 'some-mixed-string-with-spaces-underscores-and-hyphens'
5 kebab('AllThe-small Things') # 'all-the-small-things'
```

Checks if the given key exists in a dictionary.

- Use the `in` operator to check if `d` contains `key`.

```
1 def key_in_dict(d, key):
2     return (key in d)
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 key_in_dict(d, 'three') # True
```

Finds the key of the maximum value in a dictionary.

- Use `max()` with the `key` parameter set to `dict.get()` to find and return the key of the maximum value in the given dictionary.

```
1 def key_of_max(d):
2     return max(d, key = d.get)
```

```
key_of_max({'a':4, 'b':0, 'c':13}) # c
```

Finds the key of the minimum value in a dictionary.

- Use `min()` with the `key` parameter set to `dict.get()` to find and return the key of the minimum value in the given dictionary.

```
1 def key_of_min(d):
2     return min(d, key = d.get)
```

```
key_of_min({'a':4, 'b':0, 'c':13}) # b
```

Creates a flat list of all the keys in a flat dictionary.

- Use `dict.keys()` to return the keys in the given dictionary.
- Return a `list()` of the previous result.

```
1 def keys_only(flat_dict):
2     return list(flat_dict.keys())
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 keys_only(ages) # ['Peter', 'Isabel', 'Anna']
```

unlisted: true

Converts kilometers to miles.

- Follows the conversion formula `mi = km * 0.621371`.

```
1 def km_to_miles(km):
2     return km * 0.621371
```

```
km_to_miles(8.1) # 5.0331051
```

Returns the last element in a list.

- Use `lst[-1]` to return the last element of the passed list.

```
1 def last(lst):
2     return lst[-1]
```

```
last([1, 2, 3]) # 3
```

Returns the least common multiple of a list of numbers.

- Use `functools.reduce()`, `math.gcd()` and `lcm(x,y) = x * y / gcd(x,y)` over the given list.

```
1 from functools import reduce
2 from math import gcd
3
4 def lcm(numbers):
5     return reduce((lambda x, y: int(x * y / gcd(x, y))), numbers)
```

```
1 lcm([12, 7]) # 84
2 lcm([1, 3, 4, 5]) # 60
```

Takes any number of iterable objects or objects with a length property and returns the longest one.

- Use `max()` with `len()` as the `key` to return the item with the greatest length.
- If multiple objects have the same length, the first one will be returned.

```
1 def longest_item(*args):
2     return max(args, key = len)
```

```
1 longest_item('this', 'is', 'a', 'testcase') # 'testcase'
2 longest_item([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]) # [1, 2, 3, 4, 5]
3 longest_item([1, 2, 3], 'foobar') # 'foobar'
```

Maps the values of a list to a dictionary using a function, where the key-value pairs consist of the original value as the key and the result of the function as the value.

- Use `map()` to apply `fn` to each value of the list.
- Use `zip()` to pair original values to the values produced by `fn`.
- Use `dict()` to return an appropriate dictionary.

```
1 def map_dictionary(itr, fn):
2     return dict(zip(itr, map(fn, itr)))
```

```
map_dictionary([1, 2, 3], lambda x: x * x) # { 1: 1, 2: 4, 3: 9 }
```

Creates a dictionary with the same keys as the provided dictionary and values generated by running the provided function for each value.

- Use `dict.items()` to iterate over the dictionary, assigning the values produced by `fn` to each key of a new dictionary.

```
1 def map_values(obj, fn):
2     return dict((k, fn(v)) for k, v in obj.items())
```

```
1 users = {
2     'fred': { 'user': 'fred', 'age': 40 },
3     'pebbles': { 'user': 'pebbles', 'age': 1 }
4 }
5 map_values(users, lambda u : u['age']) # {'fred': 40, 'pebbles': 1}
```

Returns the maximum value of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `max()` to return the maximum value.

```
1 def max_by(lst, fn):
2     return max(map(fn, lst))
```

```
max_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 8
```

Returns the index of the element with the maximum value in a list.

- Use `max()` and `list.index()` to get the maximum value in the list and return its index.

```
1 def max_element_index(arr):
2     return arr.index(max(arr))
```

```
max_element_index([5, 8, 9, 7, 10, 3, 0]) # 4
```

Returns the `n` maximum elements from the provided list.

- Use `sorted()` to sort the list.
- Use slice notation to get the specified number of elements.
- Omit the second argument, `n`, to get a one-element list.
- If `n` is greater than or equal to the provided list's length, then return the original list (sorted in descending order).

```
1 def max_n(lst, n = 1):
2     return sorted(lst, reverse = True)[:n]
```

```
1 max_n([1, 2, 3]) # [3]
2 max_n([1, 2, 3], 2) # [3, 2]
```

Finds the median of a list of numbers.

- Sort the numbers of the list using `list.sort()`.
- Find the median, which is either the middle element of the list if the list length is odd or the average of the two middle elements if the list length is even.
- `statistics.median()` provides similar functionality to this snippet.

```

1 def median(list):
2     list.sort()
3     list_length = len(list)
4     if list_length % 2 == 0:
5         return (list[int(list_length / 2) - 1] + list[int(list_length / 2)]) / 2
6     return float(list[int(list_length / 2)])

```

```

1 median([1, 2, 3]) # 2.0
2 median([1, 2, 3, 4]) # 2.5

```

Merges two or more lists into a list of lists, combining elements from each of the input lists based on their positions.

- Use `max()` combined with a list comprehension to get the length of the longest list in the arguments.
- Use `range()` in combination with the `max_length` variable to loop as many times as there are elements in the longest list.
- If a list is shorter than `max_length`, use `fill_value` for the remaining items (defaults to `None`).
- `zip()` and `itertools.zip_longest()` provide similar functionality to this snippet.

```

1 def merge(*args, fill_value = None):
2     max_length = max([len(lst) for lst in args])
3     result = []
4     for i in range(max_length):
5         result.append([
6             args[k][i] if i < len(args[k]) else fill_value for k in range(len(args))
7         ])
8     return result

```

```

1 merge(['a', 'b'], [1, 2], [True, False]) # [[['a', 1, True], ['b', 2, False]]
2 merge(['a'], [1, 2], [True, False]) # [[['a', 1, True], [None, 2, False]]
3 merge(['a'], [1, 2], [True, False], fill_value = '_')
4 # [[[a', 1, True], ['_', 2, False]]

```

Merges two or more dictionaries.

- Create a new `dict` and loop over `dicts`, using `dictionary.update()` to add the key-value pairs from each one to the result.

```

1 def merge_dictionaries(*dicts):
2     res = dict()
3     for d in dicts:
4         res.update(d)
5     return res

```

```

1 ages_one = {
2     'Peter': 10,
3     'Isabel': 11,
4 }
5 ages_two = {
6     'Anna': 9

```

```
7 }
8 merge_dictionaries(ages_one, ages_two)
9 # { 'Peter': 10, 'Isabel': 11, 'Anna': 9 }
```

unlisted: true

Converts miles to kilometers.

- Follows the conversion formula $\text{km} = \text{mi} * 1.609344$.

```
1 def miles_to_km(miles):
2     return miles * 1.609344
```

```
miles_to_km(5.03) # 8.09500032
```

Returns the minimum value of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `min()` to return the minimum value.

```
1 def min_by(lst, fn):
2     return min(map(fn, lst))
```

```
min_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 2
```

Returns the index of the element with the minimum value in a list.

- Use `min()` and `list.index()` to obtain the minimum value in the list and then return its index.

```
1 def min_element_index(arr):
2     return arr.index(min(arr))
```

```
min_element_index([3, 5, 2, 6, 10, 7, 9]) # 2
```

Returns the `n` minimum elements from the provided list.

- Use `sorted()` to sort the list.
- Use slice notation to get the specified number of elements.
- Omit the second argument, `n`, to get a one-element list.
- If `n` is greater than or equal to the provided list's length, then return the original list (sorted in ascending order).

```
1 def min_n(lst, n = 1):
2     return sorted(lst, reverse = False)[:n]
```

```
1 min_n([1, 2, 3]) # [1]
2 min_n([1, 2, 3], 2) # [1, 2]
```

Calculates the month difference between two dates.

- Subtract `start` from `end` and use `datetime.timedelta.days` to get the day difference.
- Divide by `30` and use `math.ceil()` to get the difference in months (rounded up).

```
1 from math import ceil
2
3 def months_diff(start, end):
4     return ceil((end - start).days / 30)
```

```
1 from datetime import date
2
3 months_diff(date(2020, 10, 28), date(2020, 11, 25)) # 1
```

Returns the most frequent element in a list.

- Use `set()` to get the unique values in `lst`.
- Use `max()` to find the element that has the most appearances.

```
1 def most_frequent(lst):
2     return max(set(lst), key = lst.count)
```

```
most_frequent([1, 2, 1, 2, 3, 2, 1, 4, 2]) #2
```

Generates a string with the given string value repeated `n` number of times.

- Repeat the string `n` times, using the `*` operator.

```
1 def n_times_string(s, n):
2     return (s * n)
```

```
n_times_string('py', 4) #'pypypypy'
```

Checks if the provided function returns `True` for at least one element in the list.

- Use `all()` and `fn` to check if `fn` returns `False` for all the elements in the list.

```
1 def none(lst, fn = lambda x: x):
2     return all(not fn(x) for x in lst)
```

```
1 none([0, 1, 2, 0], lambda x: x >= 2) # False
2 none([0, 0, 0]) # True
```

Maps a number from one range to another range.

- Return `num` mapped between `outMin - outMax` from `inMin - inMax`.

```
1 def num_to_range(num, inMin, inMax, outMin, outMax):
2     return outMin + (float(num - inMin) / float(inMax - inMin)) * (outMax
3                           - outMin)
```

```
num_to_range(5, 0, 10, 0, 100) # 50.0
```

Moves the specified amount of elements to the end of the list.

- Use slice notation to get the two slices of the list and combine them before returning.

```
1 def offset(lst, offset):
2     return lst[offset:] + lst[:offset]
```

```
1 offset([1, 2, 3, 4, 5], 2) # [3, 4, 5, 1, 2]
2 offset([1, 2, 3, 4, 5], -2) # [4, 5, 1, 2, 3]
```

Pads a string on both sides with the specified character, if it's shorter than the specified length.

- Use `str.ljust()` and `str.rjust()` to pad both sides of the given string.
- Omit the third argument, `char`, to use the whitespace character as the default padding character.

```
1 from math import floor
2
3 def pad(s, length, char = ' '):
4     return s.rjust(floor((len(s) + length)/2), char).ljust(length, char)
```

```
1 pad('cat', 8) #   cat   '
2 pad('42', 6, '0') # '0004200'
3 pad('foobar', 3) # 'foobar'
```

Pads a given number to the specified length.

- Use `str.zfill()` to pad the number to the specified length, after converting it to a string.

```
1 def pad_number(n, l):
2     return str(n).zfill(l)
```

```
pad_number(1234, 6); # '001234'
```

Checks if the given string is a palindrome.

- Use `str.lower()` and `re.sub()` to convert to lowercase and remove non-alphanumeric characters from the given string.
- Then, compare the new string with its reverse, using slice notation.

```
1 from re import sub
2
3 def palindrome(s):
4     s = sub('[\W_]', '', s.lower())
5     return s == s[::-1]
```

```
palindrome('taco cat') # True
```

Converts a list of dictionaries into a list of values corresponding to the specified `key`.

- Use a list comprehension and `dict.get()` to get the value of `key` for each dictionary in `lst`.

```
1 def pluck(lst, key):
2     return [x.get(key) for x in lst]
```

```
1 simpsons = [
2     { 'name': 'lisa', 'age': 8 },
3     { 'name': 'homer', 'age': 36 },
4     { 'name': 'marge', 'age': 34 },
5     { 'name': 'bart', 'age': 10 }
6 ]
7 pluck(simpsons, 'age') # [8, 36, 34, 10]
```

Returns the powerset of a given iterable.

- Use `list()` to convert the given value to a list.
- Use `range()` and `itertools.combinations()` to create a generator that returns all subsets.
- Use `itertools.chain.from_iterable()` and `list()` to consume the generator and return a list.

```
1 from itertools import chain, combinations
2
3 def powerset(iterable):
4     s = list(iterable)
5     return list(chain.from_iterable(combinations(s, r) for r in range(len(s)+1)))
```

```
powerset([1, 2]) # [(), (1,), (2,), (1, 2)]
```

Converts an angle from radians to degrees.

- Use `math.pi` and the radian to degree formula to convert the angle from radians to degrees.

```

1 from math import pi
2
3 def rads_to_degrees(rad):
4     return (rad * 180.0) / pi

```

```

1 from math import pi
2
3 rads_to_degrees(pi / 2) # 90.0

```

Reverses a list or a string.

- Use slice notation to reverse the list or string.

```

1 def reverse(itr):
2     return itr[::-1]

```

```

1 reverse([1, 2, 3]) # [3, 2, 1]
2 reverse('snippet') # 'teppins'

```

Reverses a number.

- Use `str()` to convert the number to a string, slice notation to reverse it and `str.replace()` to remove the sign.
- Use `float()` to convert the result to a number and `math.copysign()` to copy the original sign.

```

1 from math import copysign
2
3 def reverse_number(n):
4     return copysign(float(str(n)[::-1].replace('-', '')), n)

```

```

1 reverse_number(981) # 189
2 reverse_number(-500) # -5
3 reverse_number(73.6) # 6.37
4 reverse_number(-5.23) # -32.5

```

Converts the values of RGB components to a hexadecimal color code.

- Create a placeholder for a zero-padded hexadecimal value using `'{:02X}'` and copy it three times.
- Use `str.format()` on the resulting string to replace the placeholders with the given values.

```

1 def rgb_to_hex(r, g, b):
2     return ('{:02X}' * 3).format(r, g, b)

```

```
rgb_to_hex(255, 165, 1) # 'FFA501'
```

Moves the specified amount of elements to the start of the list.

- Use slice notation to get the two slices of the list and combine them before returning.

```
1 def roll(lst, offset):
2     return lst[-offset:] + lst[:-offset]
```

```
1 roll([1, 2, 3, 4, 5], 2) # [4, 5, 1, 2, 3]
2 roll([1, 2, 3, 4, 5], -2) # [3, 4, 5, 1, 2]
```

Returns a random element from a list.

- Use `random.choice()` to get a random element from `lst`.

```
1 from random import choice
2
3 def sample(lst):
4     return choice(lst)
```

```
sample([3, 7, 9, 11]) # 9
```

Randomizes the order of the values of an list, returning a new list.

- Uses the [Fisher-Yates algorithm](#) to reorder the elements of the list.
- `random.shuffle` provides similar functionality to this snippet.

```
1 from copy import deepcopy
2 from random import randint
3
4 def shuffle(lst):
5     temp_lst = deepcopy(lst)
6     m = len(temp_lst)
7     while (m):
8         m -= 1
9         i = randint(0, m)
10        temp_lst[m], temp_lst[i] = temp_lst[i], temp_lst[m]
11    return temp_lst
```

```
1 foo = [1, 2, 3]
2 shuffle(foo) # [2, 3, 1], foo = [1, 2, 3]
```

Returns a list of elements that exist in both lists.

- Use a list comprehension on `a` to only keep values contained in both lists.

```
1 def similarity(a, b):
2     return [item for item in a if item in b]
```

```
similarity([1, 2, 3], [1, 2, 4]) # [1, 2]
```

Converts a string to a URL-friendly slug.

- Use `str.lower()` and `str.strip()` to normalize the input string.
- Use `re.sub()` to replace spaces, dashes and underscores with `-` and remove special characters.

```
1 import re
2
3 def slugify(s):
4     s = s.lower().strip()
5     s = re.sub(r'^[\w\s-]', '', s)
6     s = re.sub(r'[\s_-]+', '-', s)
7     s = re.sub(r'^-+|-+$', '', s)
8
9     return s
```

```
slugify('Hello World!') # 'hello-world'
```

Converts a string to snake case.

- Use `re.sub()` to match all words in the string, `str.lower()` to lowercase them.
- Use `re.sub()` to replace any `-` characters with spaces.
- Finally, use `str.join()` to combine all words using `-` as the separator.

```
1 from re import sub
2
3 def snake(s):
4     return '_'.join(
5         sub('([A-Z][a-z]+)', r' \1',
6             sub('([A-Z]+)', r' \1',
7                 s.replace('-', ' '))).split()).lower()
```

```
1 snake('camelCase') # 'camel_case'
2 snake('some text') # 'some_text'
3 snake('some-mixed_string With spaces_underscores-and-hyphens')
4 # 'some_mixed_string_with_spaces_underscores_and_hyphens'
5 snake('AllThe-small Things') # 'all_the_small_things'
```

Checks if the provided function returns `True` for at least one element in the list.

- Use `any()` in combination with `map()` to check if `fn` returns `True` for any element in the list.

```
1 def some(lst, fn = lambda x: x):
2     return any(map(fn, lst))
```

```
1 some([0, 1, 2, 0], lambda x: x >= 2) # True
2 some([0, 0, 1, 0]) # True
```

Sorts one list based on another list containing the desired indexes.

- Use `zip()` and `sorted()` to combine and sort the two lists, based on the values of `indexes`.
- Use a list comprehension to get the first element of each pair from the result.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the third argument.

```
1 def sort_by_indexes(lst, indexes, reverse=False):
2     return [val for (_, val) in sorted(zip(indexes, lst), key=lambda x: \
3                                         x[0], reverse=reverse)]
```

```
1 a = ['eggs', 'bread', 'oranges', 'jam', 'apples', 'milk']
2 b = [3, 2, 6, 4, 1, 5]
3 sort_by_indexes(a, b) # ['apples', 'bread', 'eggs', 'jam', 'milk', 'oranges']
4 sort_by_indexes(a, b, True)
5 # ['oranges', 'milk', 'jam', 'eggs', 'bread', 'apples']
```

Sorts the given dictionary by key.

- Use `dict.items()` to get a list of tuple pairs from `d` and sort it using `sorted()`.
- Use `dict()` to convert the sorted list back to a dictionary.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the second argument.

```
1 def sort_dict_by_key(d, reverse = False):
2     return dict(sorted(d.items(), reverse = reverse))
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 sort_dict_by_key(d) # {'five': 5, 'four': 4, 'one': 1, 'three': 3, 'two': 2}
3 sort_dict_by_key(d, True)
4 # {'two': 2, 'three': 3, 'one': 1, 'four': 4, 'five': 5}
```

Sorts the given dictionary by value.

- Use `dict.items()` to get a list of tuple pairs from `d` and sort it using a lambda function and `sorted()`.
- Use `dict()` to convert the sorted list back to a dictionary.
- Use the `reverse` parameter in `sorted()` to sort the dictionary in reverse order, based on the second argument.
- **⚠️ NOTICE:** Dictionary values must be of the same type.

```
1 def sort_dict_by_value(d, reverse = False):
2     return dict(sorted(d.items(), key = lambda x: x[1], reverse = reverse))
```

```
1 d = {'one': 1, 'three': 3, 'five': 5, 'two': 2, 'four': 4}
2 sort_dict_by_value(d) # {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
3 sort_dict_by_value(d, True)
4 # {'five': 5, 'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

Splits a multiline string into a list of lines.

- Use `str.split()` and `'\n'` to match line breaks and create a list.
- `str.splitlines()` provides similar functionality to this snippet.

```
1 def split_lines(s):
2     return s.split('\n')
```

```
1 split_lines('This\nis a\nmultiline\nstring.\n')
2 # ['This', 'is a', 'multiline', 'string.', '']
```

Flattens a list, by spreading its elements into a new list.

- Loop over elements, use `list.extend()` if the element is a list, `list.append()` otherwise.

```
1 def spread(arg):
2     ret = []
3     for i in arg:
4         ret.extend(i) if isinstance(i, list) else ret.append(i)
5     return ret
```

```
spread([1, 2, 3, [4, 5, 6], [7], 8, 9]) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Calculates the sum of a list, after mapping each element to a value using the provided function.

- Use `map()` with `fn` to map each element to a value using the provided function.
- Use `sum()` to return the sum of the values.

```
1 def sum_by(lst, fn):
2     return sum(map(fn, lst))
```

```
sum_by([{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }], lambda v : v['n']) # 20
```

Returns the sum of the powers of all the numbers from `start` to `end` (both inclusive).

- Use `range()` in combination with a list comprehension to create a list of elements in the desired range raised to the given `power`.
- Use `sum()` to add the values together.
- Omit the second argument, `power`, to use a default power of `2`.
- Omit the third argument, `start`, to use a default starting value of `1`.

```
1 def sum_of_powers(end, power = 2, start = 1):
2     return sum([(i) ** power for i in range(start, end + 1)])
```

```
1 sum_of_powers(10) # 385
2 sum_of_powers(10, 3) # 3025
3 sum_of_powers(10, 3, 5) # 2925
```

Returns the symmetric difference between two iterables, without filtering out duplicate values.

- Create a `set` from each list.
- Use a list comprehension on each of them to only keep values not contained in the previously created set of the other.

```
1 def symmetric_difference(a, b):
2     (_a, _b) = (set(a), set(b))
3     return [item for item in a if item not in _b] + [item for item in b
4             if item not in _a]
```

```
symmetric_difference([1, 2, 3], [1, 2, 4]) # [3, 4]
```

Returns the symmetric difference between two lists, after applying the provided function to each list element of both.

- Create a `set` by applying `fn` to each element in every list.
- Use a list comprehension in combination with `fn` on each of them to only keep values not contained in the previously created set of the other.

```
1 def symmetric_difference_by(a, b, fn):
2     (_a, _b) = (set(map(fn, a)), set(map(fn, b)))
3     return [item for item in a if fn(item) not in _b] + [item
4             for item in b if fn(item) not in _a]
```

```
1 from math import floor
2
3 symmetric_difference_by([2.1, 1.2], [2.3, 3.4], floor) # [1.2, 3.4]
```

Returns all elements in a list except for the first one.

- Use slice notation to return the last element if the list's length is more than `1`.
- Otherwise, return the whole list.

```
1 def tail(lst):
2     return lst[1:] if len(lst) > 1 else lst
```

```
1 tail([1, 2, 3]) # [2, 3]
2 tail([1]) # [1]
```

Returns a list with `n` elements removed from the beginning.

- Use slice notation to create a slice of the list with `n` elements taken from the beginning.

```
1 def take(itr, n = 1):
2     return itr[:n]
```

```
1 take([1, 2, 3], 5) # [1, 2, 3]
2 take([1, 2, 3], 0) # []
```

Returns a list with `n` elements removed from the end.

- Use slice notation to create a slice of the list with `n` elements taken from the end.

```
1 def take_right(itr, n = 1):
2     return itr[-n:]
```

```
1 take_right([1, 2, 3], 2) # [2, 3]
2 take_right([1, 2, 3]) # [3]
```

Returns the binary representation of the given number.

- Use `bin()` to convert a given decimal number into its binary equivalent.

```
1 def to_binary(n):
2     return bin(n)
```

```
to_binary(100) # 0b1100100
```

Combines two lists into a dictionary, where the elements of the first one serve as the keys and the elements of the second one serve as the values.

The values of the first list need to be unique and hashable.

- Use `zip()` in combination with `dict()` to combine the values of the two lists into a dictionary.

```
1 def to_dictionary(keys, values):
2     return dict(zip(keys, values))
```

```
to_dictionary(['a', 'b'], [1, 2]) # { a: 1, b: 2 }
```

Returns the hexadecimal representation of the given number.

- Use `hex()` to convert a given decimal number into its hexadecimal equivalent.

```
1 def to_hex(dec):
2     return hex(dec)
```

```
1 to_hex(41) # 0x29
2 to_hex(332) # 0x14c
```

Converts a date to its ISO-8601 representation.

- Use `datetime.datetime.isoformat()` to convert the given `datetime.datetime` object to an ISO-8601 date.

```
1 from datetime import datetime
2
3 def to_iso_date(d):
4     return d.isoformat()
```

```
1 from datetime import datetime
2
3 to_iso_date(datetime(2020, 10, 25)) # 2020-10-25T00:00:00
```

Converts an integer to its roman numeral representation.

Accepts value between `1` and `3999` (both inclusive).

- Create a lookup list containing tuples in the form of (roman value, integer).
- Use a `for` loop to iterate over the values in `lookup`.
- Use `divmod()` to update `num` with the remainder, adding the roman numeral representation to the result.

```
1 def to_roman_numeral(num):
2     lookup = [
3         (1000, 'M'),
4         (900, 'CM'),
5         (500, 'D'),
6         (400, 'CD'),
7         (100, 'C'),
8         (90, 'XC'),
9         (50, 'L'),
10        (40, 'XL'),
11        (10, 'X'),
12        (9, 'IX'),
13        (5, 'V'),
14        (4, 'IV'),
15        (1, 'I'),
16    ]
17    res = ''
18    for (n, roman) in lookup:
19        (d, num) = divmod(num, n)
20        res += roman * d
21    return res
```

```
1 to_roman_numeral(3) # 'III'
2 to_roman_numeral(11) # 'XI'
3 to_roman_numeral(1998) # 'MCMXCVIII'
```

Transposes a two-dimensional list.

- Use `*lst` to get the provided list as tuples.
- Use `zip()` in combination with `list()` to create the transpose of the given two-dimensional list.

```
1 def transpose(lst):
2     return list(zip(*lst))
```

```
1 transpose([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
2 # [(1, 4, 7, 10), (2, 5, 8, 11), (3, 6, 9, 12)]
```

Builds a list, using an iterator function and an initial seed value.

- The iterator function accepts one argument (`seed`) and must always return a list with two elements ([`value`, `nextSeed`]) or `False` to terminate.
- Use a generator function, `fn_generator`, that uses a `while` loop to call the iterator function and `yield` the `value` until it returns `False`.
- Use a list comprehension to return the list that is produced by the generator, using the iterator function.

```
1 def unfold(fn, seed):
2     def fn_generator(val):
3         while True:
4             val = fn(val[1])
5             if val == False: break
6             yield val[0]
7     return [i for i in fn_generator([None, seed])]
```

```
1 f = lambda n: False if n > 50 else [-n, n + 10]
2 unfold(f, 10) # [-10, -20, -30, -40, -50]
```

Returns every element that exists in any of the two lists once.

- Create a `set` with all values of `a` and `b` and convert to a `list`.

```
1 def union(a, b):
2     return list(set(a + b))
```

```
union([1, 2, 3], [4, 3, 2]) # [1, 2, 3, 4]
```

Returns every element that exists in any of the two lists once, after applying the provided function to each element of both.

- Create a `set` by applying `fn` to each element in `a`.
- Use a list comprehension in combination with `fn` on `b` to only keep values not contained in the previously created set, `_a`.
- Finally, create a `set` from the previous result and `a` and transform it into a `list`.

```
1 def union_by(a, b, fn):
```

```
2     _a = set(map(fn, a))
3     return list(set(a + [item for item in b if fn(item) not in _a]))
```

```
1 from math import floor
2
3 union_by([2.1], [1.2, 2.3], floor) # [2.1, 1.2]
```

Returns the unique elements in a given list.

- Create a `set` from the list to discard duplicated values, then return a `list` from it.

```
1 def unique_elements(li):
2     return list(set(li))
```

```
unique_elements([1, 2, 2, 3, 4, 3]) # [1, 2, 3, 4]
```

Returns a flat list of all the values in a flat dictionary.

- Use `dict.values()` to return the values in the given dictionary.
- Return a `list()` of the previous result.

```
1 def values_only(flat_dict):
2     return list(flat_dict.values())
```

```
1 ages = {
2     'Peter': 10,
3     'Isabel': 11,
4     'Anna': 9,
5 }
6 values_only(ages) # [10, 11, 9]
```

Returns the weighted average of two or more numbers.

- Use `sum()` to sum the products of the numbers by their weight and to sum the weights.
- Use `zip()` and a list comprehension to iterate over the pairs of values and weights.

```
1 def weighted_average(nums, weights):
2     return sum(x * y for x, y in zip(nums, weights)) / sum(weights)
```

```
weighted_average([1, 2, 3], [0.6, 0.2, 0.3]) # 1.72727
```

Tests a value, `x`, against a testing function, conditionally applying a function.

- Check if the value of `predicate(x)` is `True` and if so return `when_true(x)`, otherwise return `x`.

```
1 def when(predicate, when_true):
2     return lambda x: when_true(x) if predicate(x) else x
```

```
1 double_even_numbers = when(lambda x: x % 2 == 0, lambda x : x * 2)
2 double_even_numbers(2) # 4
3 double_even_numbers(1) # 1
```

Converts a given string into a list of words.

- Use `re.findall()` with the supplied `pattern` to find all matching substrings.
- Omit the second argument to use the default regexp, which matches alphanumeric and hyphens.

```
1 import re
2
3 def words(s, pattern = '[a-zA-Z-]+'):
4     return re.findall(pattern, s)
```

```
1 words('I love Python!!!') # ['I', 'love', 'Python']
2 words('python, javaScript & coffee') # ['python', 'javaScript', 'coffee']
3 words('build -q --out one-item', r'\b[a-zA-Z-]+\b')
4 # ['build', 'q', 'out', 'one-item']
```

Interview Prep

Interview Resources

 DS-Algo-Codebase

<https://bgoonz-branch-the-algos.vercel.app/>

 GitHub - bgoonz/INTERVIEW-PREP-COMPLETE: INTERVIEW-PREP-ARCHIVE

<https://github.com/bgoonz/INTERVIEW-PREP-COMPLETE>

 README

<https://determined-dijkstra-ee7390.netlify.app/>

 idk - CodeSandbox

<https://codesandbox.io/s/idk-q48t0>

Unsorted Examples

Design your implementation of the linked list. You can choose to use the singly linked list or the doubly linked list.

A node in a singly linked list should have two attributes: `val`

and

`next`. `val` is the value of the current node

`next` is a pointer/reference to the next node.

If you want to use the doubly linked list, you will need one more attribute `prev` to indicate the previous node in the linked list.

Assume all nodes in the linked list are 0-indexed.

Implement these functions in your linked list class:

 Loading...

<https://leetcode.com/problems/design-linked-list>

```
1  class ListNode:
2      def __init__(self, val=None):
3          self.val = val
4          self.next = None
5
6
7
8  class MyLinkedList:
9      def __init__(self):
10         """
11             Initialize your data structure here.
12         """
13         self.head = None
14
15     def get(self, index: int) -> int:
16         """
17             Get the value of the index-th node in the linked list. If the index is invalid, return -1.
18         """
19         if index >= 0 and index <= 1000:
20             counter = 0
21             node = self.head
22             while node:
23                 if counter == index:
24                     return node.val
25                 node = node.next
26                 counter += 1
27
28     def addAtHead(self, val: int) -> None:
29         """
30             Add a node of value val before the first element of the linked list. After the insertion, the new node will
31             be at the head of the list.
32         """
33         if self.head != None:
34             newnode = ListNode(val)
35             newnode.next = self.head
36             self.head = newnode
```

```

36         self.head = newnode
37     else:
38         self.head = ListNode(val)
39
40     def addAtTail(self, val: int) -> None:
41         """
42             Append a node of value val to the last element of the linked list.
43             """
44         if self.head != None:
45             newnode = ListNode(val)
46             node = self.head
47             while node:
48                 if node.next == None:
49                     node.next = newnode
50                     break
51                 node = node.next
52             else:
53                 self.head = ListNode(val)
54
55     def addAtIndex(self, index: int, val: int) -> None:
56
57     # Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list,
58
59     if index >= 0 and index <= 1000:
60         node = self.head
61         counter = 1
62         if index == 0:
63             if self.head != None:
64                 newnode = ListNode(val)
65                 newnode.next = self.head
66                 self.head = newnode
67             else:
68                 self.head = ListNode(val)
69         else:
70             while node:
71                 if index == counter:
72                     newnode = ListNode(val)
73                     temp = node.next
74                     node.next = newnode
75                     newnode.next = temp
76                     break
77                 node = node.next
78                 counter += 1
79         else:
80             pass
81
82     def deleteAtIndex(self, index: int) -> None:
83
84     #Delete the index-th node in the linked list, if the index is valid.
85
86     node = self.head
87     counter = 1
88     if node.next != None:
89         if index == 0:
90             temp = node.next
91             self.head = temp
92         else:
93             while node:
94                 if index == counter and node.next != None:
95                     node.next = node.next.next
96                     break
97
98             node = node.next
99             counter += 1

```

How to Write an Effective Resume of Python Developer

How to Write an Effective Resume of Python Developer

With the world's orientation towards digital technology, The Python programming language has raised its value. It has overshadowed many occupations and Python developers have numerous opportunities for career growth.

With knowledge of Python programming language, you can work in innovative fields such as artificial intelligence, machine learning, and data science. However, to get there you first need to gain the attention of employers. That's when the resume steps in.

A resume presents the first impression that hiring managers make of you. How you write your resume will determine whether you'll have a chance to win them over at the interview or you'll end up in the "don't contact" pile. If you are ready to get some job interviews scheduled, here are the tips you should follow when writing your resume.

Customize the Resume to Job Description

While job descriptions usually all sound pretty much the same, you should customize your resume. There is always a way to make your resume more suitable for individual job positions. Instead of typical generalization, adapt your resume before you click the send button.

Start with creating a template with all the basic information that all employers require. Later, you can frame the resume based on specifications that different companies seek for. For example, if an employer requires at least two years of experience, your resume must contain specific job experiences that show that you have fulfilled that condition. Otherwise, don't waste time on that application. One of the biggest mistakes that many job applicants make is including their job experience that has nothing to do with the occupation they pursue. The fact that you worked at McDonald's when you were sixteen won't influence the employer to give you the job of a Python developer.

Choose the Layout Based on Your Experience

The dilemma that worries many resume-makers is how to form the resume. When choosing a layout for a Python developer resume you will encounter three options:

1. Chronological layout – Lists your experiences in chronological order
2. Reverse chronological layout – Puts the focus on the relevant experiences like a timeline (starting from the last job position)
3. Functional layout – Emphasizes your skills

The most common form is the chronological layout. However, what makes reverse chronological layout increasingly popular is that it highlights your most recent experience. This works best if your last job position was a Python developer as that would instantly show the employer that you have experience.

The functional layout can be tricky as it demands creativity. The situation in which this format is a good choice is when you don't have real work experience but you do have strong skills that depict you as a promising developer.

Keep it Concise

The length of the resume has always been a troublesome topic. People usually aim to make the resume longer so that it seems like they have more experience. That's not such a good idea.

A national survey conducted by [Harris Poll for CareerBuilder](#) showed that concise resumes have more than an 80% chance of being accepted. A two-page resume is claimed to have the perfect length.

Presenting yourself in a concise form will keep the reader's attention until the end of the resume. Keep in mind that hiring managers go through at least dozens of resumes and they won't have the concentration or desire to read essay-like CVs.

If you can't manage to present the information concisely, you can always turn to writing and editing professionals. For example, experts that work for [BestEssayEducation](#) can rephrase some points for a cleaner look. Or, you can use editing tools like [HemingwayEditor](#).

Avoid Cliché Terms

Did you know that clichéd and generic terms that hiring managers have seen in almost all resumes increase your chance of not getting the job by 50%? Companies that look for Python developers who are experts in their job and great team players won't be drawn to dull and typical resumes.

Luckily, [New College of the Humanities](#) has researched top cliché terms that you shouldn't use in your resume. They surveyed 2,000 employers and the terms that came up as repelling are the following:

- Works well under pressure
- Excellent written communication skills
- Can work independently
- Problem solver
- Hard worker
- Good communicator
- Proactive
- Enthusiastic
- Team player
- Good listener

A generic resume will hurt your possibility to stand out and attract the attention of potential employers.

Pay Attention to Readability

According to [research](#), resumes that have typos or bad grammar have the highest chance of getting instantly rejected (77%). Meaning, that your expertise can fall in the shadow of a poorly written resume.

Even though you are applying for a position of a Python developer, that doesn't mean that you can be careless. If you want to ensure that no mistakes pass you by, you can use some of these services and tools:

- [TrustMyPaper](#) – This writing service only works with most talented writers with great attention to detail. They can signal any mistakes or inconsistencies in your resume.
- [GrabMyEssay](#) – With both writing and editing services at your disposal, you can find all that you need for polishing your essay on this website.
- [Studicus](#) – The numerous positive testimonials speak for Studicus' professionalism. Writers must have years of experience to work for this company so you'll be teamed up with experts.
- [Grammarly](#) – If you are looking for a quick fix, Grammarly is an online editing tool that will ensure that your resume doesn't have any grammar or spelling mistakes.
- [Readable](#) – Readability checkers such as Readable will point out any confusing or ambiguous parts of your resume.

Don't Go Overboard with the Design

The first thing that hiring managers will notice about your resume is the design. Using a flashy and chaotic design doesn't really say "this is the best Python developer for this company." Simplicity is the safest and most elegant choice when it comes to design. Keep it simple and consistent with the style and you won't have to worry about whether the design will send the wrong message.

- Here are a few suggestions when it comes to resume design:
- Underline headers and sections
- Use fonts that are easy to read (Times New Roman, Arial, etc.)
- The recommendable font size is 12 (or 11 if you want to fit information within 2 pages)
- Use bullets for listing
- Use bold text for emphasis (for job titles for example)

- List qualifications and skills with bullets rather than stating them in a paragraph

Final Thoughts

The number of jobs for Python developers is continually growing. As companies switch to advanced technologies their need for Python programming language appears. This is the perfect time to find the job of your dreams. But first, consider the above-mentioned tips and create a winning resume.

Kristin Savage nourishes, sparks, and empowers using the magic of a word. Along with pursuing her degree in Creative Writing, Kristin was gaining experience in the publishing industry, with expertise in marketing strategy for publishers and authors. Besides working as a freelance writer at [WowGrade](#) she also does some editing work at [SupremeDissertations](#). In her free time, Kristin likes to travel and explore new countries around the world.

Interview Checklist

150 Practice Problems & Solutions

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 :*

```
Twinkle, twinkle, little star,      How I wonder what you are!
                                                Up above the world so high,
                                                Like a diamond in the sky.
```

[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 :*
- ```
numbers = [386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345, 399, 162, 758,
```
- [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](#) [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](#)

## Installations Setup & Env

# python-setup

## Installing Python Modules

Email

[distutils-sig@python.org](mailto:distutils-sig@python.org)

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the installation part of the process. For a guide to creating and sharing your own Python projects, refer to the [distribution guide](#).

Note

For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.

## Key terms

- `pip` is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.
- A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.
- `venv` is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing `pip` into all created virtual environments.
- `virtualenv` is a third party alternative (and predecessor) to `venv`. It allows virtual environments to be used on versions of Python prior to 3.4, which either don't provide `venv` at all, or aren't able to automatically install `pip` into created environments.
- The [Python Packaging Index](#) is a public repository of open source licensed packages made available for use by other Python users.
- the [Python Packaging Authority](#) is the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation, and issue trackers on both [GitHub](#) and [Bitbucket](#).
- `distutils` is the original build and distribution system first added to the Python standard library in 1998. While direct use of `distutils` is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).

Changed in version 3.5: The use of `venv` is now recommended for creating virtual environments.

See also

[Python Packaging User Guide: Creating and using virtual environments](#)

## Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

## Note

For POSIX users (including Mac OS X and Linux users), the examples in this guide assume the use of a [virtual environment](#).

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It's also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as `>`, `<` or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
1 python -m pip install SomePackage==1.0.4 # specific version
2 python -m pip install "SomePackage>=1.0.4" # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade SomePackage
```

More information and resources regarding `pip` and its capabilities can be found in the [Python Packaging User Guide](#).

Creation of virtual environments is done through the `venv` module. Installing packages into an active virtual environment uses the commands shown above.

## See also

[Python Packaging User Guide: Installing Python Distribution Packages](#)

## How do I ...?

These are quick answers or links for some common tasks.

### ... install `pip` in versions of Python prior to Python 3.4?

Python only started bundling `pip` with Python 3.4. For earlier versions, `pip` needs to be “bootstrapped” as described in the Python Packaging User Guide.

## See also

[Python Packaging User Guide: Requirements for Installing Packages](#)

### ... install packages just for the current user?

Passing the `--user` option to `python -m pip install` will install a package just for the current user, rather than for all users of the system.

### ... install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren't currently easy to install using `pip` directly. At this point in time, it will often be easier for users to install these packages by [other means](#) rather than attempting to install them with `pip`.

See also

[Python Packaging User Guide: Installing Scientific Packages](#)

#### **... work with multiple versions of Python installed in parallel?**

On Linux, Mac OS X, and other POSIX systems, use the versioned Python commands in combination with the `-m` switch to run the appropriate copy of `pip`:

```
1 python2 -m pip install SomePackage # default Python 2
2 python2.7 -m pip install SomePackage # specifically Python 2.7
3 python3 -m pip install SomePackage # default Python 3
4 python3.4 -m pip install SomePackage # specifically Python 3.4
```

Appropriately versioned `pip` commands may also be available.

On Windows, use the `py` Python launcher in combination with the `-m` switch:

```
1 py -2 -m pip install SomePackage # default Python 2
2 py -2.7 -m pip install SomePackage # specifically Python 2.7
3 py -3 -m pip install SomePackage # default Python 3
4 py -3.4 -m pip install SomePackage # specifically Python 3.4
```

## Common installation issues

### Installing into the system Python on Linux

On Linux systems, a Python installation will typically be included as part of the distribution. Installing into this Python installation requires root access to the system, and may interfere with the operation of the system package manager and other components of the system if a component is unexpectedly upgraded using `pip`.

On such systems, it is often better to use a virtual environment or a per-user installation when installing packages with `pip`.

### Pip not installed

It is possible that `pip` does not get installed by default. One potential fix is:

```
python -m ensurepip --default-pip
```

There are also additional resources for [installing pip](#).

### Installing binary extensions

Python has typically relied heavily on source based distribution, with end users being expected to compile extension modules from source as part of the installation process.

With the introduction of support for the binary `wheel` format, and the ability to publish wheels for at least Windows and Mac OS X through the Python Packaging Index, this problem is expected to diminish over time, as users are more regularly able to install pre-built extensions rather than needing to build them themselves.

Some of the solutions for installing [scientific software](#) that are not yet available as pre-built `wheel` files may also help with obtaining other binary extensions without needing to build them locally.

# List Of Python Cheat Sheets

 [python-cheatsheet.py](https://gist.github.com/bgoonz/dd7fd80df384fba134b42f256298bdb4)

<https://gist.github.com/bgoonz/dd7fd80df384fba134b42f256298bdb4>

## Table of Contents

- Best Python Cheat Sheets
  - (1) Python 3 Cheat Sheet
  - (2) Python Beginner Cheat Sheet
  - (3) Python for Data Science
  - (4) Python for Data Science (Importing Data)
  - (5) Python Cheatography Cheat Sheet
  - (6) The Ultimative Python Cheat Sheet Course (5x Email Series)
  - (7) Dataquest Data Science Cheat Sheet – Python Basics
  - (8) Dataquest Data Science Cheat Sheet – Intermediate
  - (9) Dataquest NumPy
  - (10) Python For Data Science (Bokeh)
  - (11) Pandas Cheat Sheet for Data Science
  - (12) Regular Expressions Cheat Sheet
  - (13) The World's Most Concise Python Cheat Sheet
- What Cheat Sheets are NOT a Replacement For
- Cheat Sheet Alternatives
  - Flashcards
  - Personalised Cheat Sheets
  - Projects
- Summary
- Related Posts

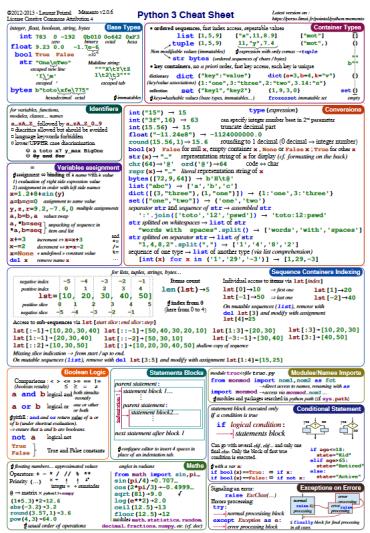
 <https://s3.amazonaws.com/dq-blog-files/python-cheat-sheet-basic.pdf>

<https://s3.amazonaws.com/dq-blog-files/python-cheat-sheet-basic.pdf>

## Best Python Cheat Sheets

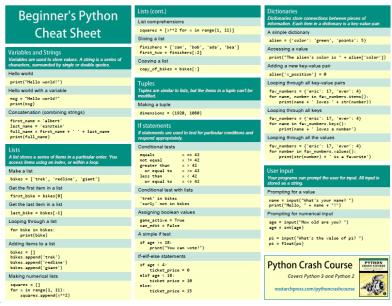
So let's dive into the best Python cheat sheets recommended by us.

### (1) Python 3 Cheat Sheet



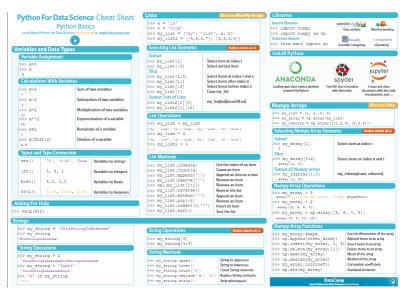
This is the best single cheat sheet. It uses every inch of the page to deliver value and covers everything you need to know to go from beginner to intermediate. Topics covered include container types, conversions, modules, maths, conditionals and formatting to name a few. A highly recommended 2-page sheet!

## (2) Python Beginner Cheat Sheet



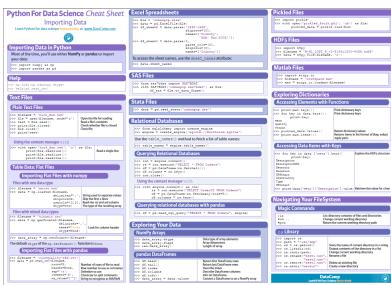
Some might think this cheat sheet is a bit lengthy. At 26 pages, it is the most comprehensive cheat sheets out there. It explains variables, data structures, exceptions, and classes – to name just a few. If you want the most thorough cheat sheet, pick this one.

## (3) Python for Data Science



Some of the most popular things to do with Python are Data Science and Machine Learning. In this cheat sheet, you will learn the basics of Python and the most important scientific library: **NumPy** (Numerical Python). You'll learn the basics plus the most important NumPy functions. If you are using Python for Data Science, download this cheat sheet.

## (4) Python for Data Science (Importing Data)



This Python data science cheat sheet from DataCamp is all about getting data into your code. Think about it: importing data is one of the most important tasks when working with data. Increasing your skills in this area will make you a better data scientist—and a better coder overall!

## (5) Python Cheatography Cheat Sheet

This image shows a detailed Python cheat sheet by Dave Child. It is divided into several sections: 'Python sys Variables', 'Python Class Special Methods', 'Python String Methods (cont.)', 'Python File Methods', 'Python List Methods', 'Python String Methods', 'Python Webkit and Slices', 'Python Date/Time Methods', and 'Python Dictionary Methods'. Each section provides a list of methods and their descriptions.

**Python sys Variables**

- args Command-line args
- builtins Module C modules
- byteorder Current byte order
- chardet\_ua Recognized user agent
- checkinterval Frequency
- exec\_prefix Root directory
- executive External executable
- exit Exit function
- modules Loaded modules
- path Search path
- platform Current platform
- stdin, stdout, stderr Standard file objects for I/O
- version\_info Python version info
- warnings Version number

**Python on argparse**

- argparse Help message
- sys.argv[1] bar
- sys.argv[2] -c
- sys.argv[3] -qip
- sys.argv[4] -n6

Usage example for the command: \$ python my\_py\_bar -c que -h

**Python on Variables**

- deep Reference exp
- osdir Current dir using
- depth Default search path
- donut Path or not device
- exists Path exists
- isroot Line separator
- name Name of file
- parent Parent dir using
- pathsep Path separator
- real Path separator
- registered OS command: "ls", "cat", "grep", "find", "tar", "tarz", "tarbz", "tarxz", "tarxz"

Published 18th October, 2011.  
Last updated 12th May, 2016.  
Page 1 of 2.

Sponsored by [AuditCloud.com](#)  
Everyone has a flaw in them. Fix it! Visit <https://auditcloud.com>

This cheat sheet is for more advanced learners. It covers class, string and list methods as well as system calls from the `sys` module. Once you're comfortable defining basic classes and command line interfaces (CLIs), get this cheat sheet. It will take you to another level.

## (6) The Ultimative Python Cheat Sheet Course (5x Email Series)

## Python Cheat Sheet - Classes

"A puzzle a day to learn, code, and play" → Visit [finxter.com](http://finxter.com)

| Description      |                                                                                                                                                                                                                                                                      | Example                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Classes</b>   | A class encapsulates data and functionality - data as attributes, and functionality as methods. It is a blueprint to create concrete instances in the memory.                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Class</b>     |                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Instances</b> |                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Instance</b>  | You are an instance of the class <code>human</code> . An instance is a concrete implementation of a class: all attributes of an instance have a fixed value. Your hair is blond, brown, or black - but never unspecified.                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|                  | Each instance has its own attributes independent of other instances. Yet, class variables are different. These are data values associated with the class, not the instances. Hence, all instances share the same class variable <code>species</code> in the example. |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Self</b>      | The first argument when defining any method is always the <code>self</code> argument. This argument specifies the instance on which you call the method.                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|                  | <code>self</code> gives the Python interpreter the information about the concrete instance. To define a method, you use <code>self</code> to modify the instance attributes. But to call an instance method, you do not need to specify <code>self</code> .          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Creation</b>  | You can create classes "on the fly" and use them as logical units to store complex data types.                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|                  | <pre>class Employee():     pass Alice = Employee() Alice.firstname = "alice" Alice.lastname = "wonderland" print(Alice.firstname + " "       + Alice.lastname + " "       + str(Alice.salary) + "\$") # alice wonderland 122000</pre>                                | <pre>class Dog:     """ Blueprint of a dog """     species = ["canis lupus"]      def __init__(self, name, color):         self.name = name         self.state = "sleeping"         self.color = color      def command(self, x):         if x == self.name:             self.bark(2)         elif x == "sit":             self.state = "sit"         else:             self.state = "wag tail"      def bark(self, freq):         for i in range(freq):             print("[" + self.name                   + "] woof!")  Bello = Dog("bello", "black") Alice = Dog("alice", "white")  print(Bello.color) # black print(Alice.color) # white  Bello.bark(1) # [bello]: Woof!  Alice.command("sit") print("[alice]: " + Alice.state) # [alice]: sit  Bello.command("no") print("[bello]: " + Bello.state) # [bello]: wag tail  Alice.command("alice") # [alice]: sit # [alice]: Woof!  Bello.species += ["wulf"] print(len(Bello.species)       == len(Alice.species)) # True (!)</pre> |

**finxter**

Want to learn Python well, but don't have much time? Then this course is for you. It contains 5 carefully designed PDF cheat sheets. Each cheat sheet takes you one step further into the rabbit hole. You will learn practical Python concepts from the hand-picked examples and code snippets. The topics include basic keywords, simple and complex data types, crucial string and list methods, and powerful Python one-liners. If you lead a busy life and do not want to compromise on quality, this is the cheat sheet course for you!



[https://s3.amazonaws.com/assets.datacamp.com/blog\\_assets/Python\\_Bokeh\\_Cheat\\_Sheet.pdf](https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Bokeh_Cheat_Sheet.pdf)

[https://s3.amazonaws.com/assets.datacamp.com/blog\\_assets/Python\\_Bokeh\\_Cheat\\_Sheet.pdf](https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Bokeh_Cheat_Sheet.pdf)

**(7) Dataquest Data Science Cheat Sheet – Python Basics**

## Data Science Cheat Sheet

Python Basics

**BASICS, PRINTING AND GETTING HELP**

```
x = 3 - Assign 3 to the variable x
print(x) - Print the value of x
type(x) - Return the type of the variable x (in this case, int for integer)
```

---

**READING FILES**

```
f = open("my_file.txt","r")
file_as_string = f.read()
Open the file my_file.txt and assign its contents to s
with open("my_file.txt","r") as f:
 file_as_string = f.read()
 # Open the CSV file my_dataset.csv and assign its data to the list of lists csv_as_list
```

---

**STRINGS**

```
s = "hello" - Assign the string "hello" to the variable s
s[0] - The first character of s
s[-1] - The last character of s
s[1:3] - The characters from index 1 to 3
s[1:-1] - The characters from index 1 to the second-to-last character
s.startswith("he") - Test whether s starts with the substring "he"
s.endswith("le") - Test whether s ends with the substring "le"
s[1:4] = "T" - Format s[1:4] with the value "T"
s.replace("e","x") - Returns a new string based on s with all occurrences of "e" replaced with "x"
s.split(" ") - Split the string s into a list of strings, separating on the character " " and return that list
```

---

**NUMERIC TYPES AND MATHEMATICAL OPERATIONS**

```
s = int("2.5") - Convert the string "2.5" to the integer 2 and assign the result to s
f = float("2.5") - Convert the string "2.5" to the float value 2.5 and assign the result to f
s + 5 - Addition
s - 5 - Subtraction
s / 2 - Division
s * 2 - Multiplication
```

---

**LISTS**

```
l = [100,21,88,3] - Assign a list containing the integers 100, 21, 88, and 3 to the variable l
l = list() - Create an empty list and assign it to l
l[0] - Return the first value in the list l
l[-1] - Return the last value in the list l
l[1:3] - Return a slice (list) containing the second and third values of l
len(l) - Return the number of elements in l
max(l) - Return the maximum value of l
min(l) - Return the minimum value from l
max(l) - Return the maximum value from l
l.append(16) - Append the value 16 to the end of l
l.sort() - Sort the items in l in ascending order
"-".join(["A", "B", "C", "D"]) - Converts the list ["A", "B", "C", "D"] into the string "A & B & C & D"
```

---

**DICTIONARIES**

```
d = {"CA": "Canada", "GB": "Great Britain",
 "IN": "India", "US": "USA"} - Create a dictionary with pairs of "CA", "GB", and "IN" and corresponding values of "Canada", "Great Britain", and "India"
d["GB"] - Return the value from the dictionary d that has the key "GB"
d.get("AU", "Sorry") - Return the value from the string "Sorry" if the key "AU" is not found in d
d.keys() - Return a list of the keys from d
d.values() - Return a list of the values from d
d.items() - Return a list of (key, value) pairs from d
```

---

**MODULES AND FUNCTIONS**

```
The body of a function is defined through indentation.
import math - Import the module math
from math import sqrt - Import the function sqrt from the module math
```

---

**def calculate(addition\_one, addition\_two, exponent\_val, factor=1):
 result = (value\_one+value\_two)\*\*exponent\*factor
 return result**

**BOOLEAN COMPARISONS**

```
x == 5 - Test whether x is equal to 5
x != 5 - Test whether x is not equal to 5
x > 5 - Test whether x is greater than 5
x < 5 - Test whether x is less than 5
x >= 5 - Test whether x is greater than or equal to 5
x <= 5 - Test whether x is less than or equal to 5
x == 5 or name == "alfred" - Test whether x is equal to 5 or name is equal to "alfred"
x != 5 and name != "alfred" - Test whether x is not equal to 5 and name is not equal to "alfred"
x in l - Checks whether the value x exists in the list l
"GB" in d - Checks whether the value "GB" exists in the keys for d
```

---

**IF STATEMENTS AND LOOPS**

```
If body of if statements and loops are defined through indentation.
if x < 5:
 print("x is greater than five", format(x))
else:
 print("x is negative", format(x))
for i in range(5):
 print("i is between zero and five", format(i))
 - Test the value of the variable x and run the code inside the loop
 for value in l:
 print(value)
 - Iterate over each value in l, running the code in the body of the loop with each iteration
 while x > 10:
 x -= 1
 - Run the code in the body of the loop until the value of x is no longer less than 10
```

The wonderful team at Dataquest have put together this comprehensive beginners cheat sheet. It covers all the basic data types, looping and reading files. It's beautifully designed and is the first of a series.

## (8) Dataquest Data Science Cheat Sheet – Intermediate

**KEY BASICS, PRINTING AND GETTING HELP**

This cheat sheet assumes you are familiar with the content of our Python Basics Cheat Sheet

- A Python string variable
- A Python integer variable
- A Python float variable
- A Python dictionary variable

---

**LISTS**

```
l.pop(0) - Returns the fourth item from 1 and deletes it from the list
l.remove(x) - Removes the first item in l that is equal to x
l.reverse() - Reverses the order of the items in l
l[1:-1] - Returns every second item from l, commencing from the 1st item
l[-1:] - Returns the last 5 items from l specific axis
```

---

**STRINGS**

```
s.lower() - Returns a lowercase version of s
s.title() - Returns s with the first letter of every word capitalized
"23".rfill(4) - Returns "0023" by left-filling the string with 0's to make it's length 4.
s.splitlines() - Returns a sequence by splitting the string on the newline characters
Python strings share some common methods with lists
s[1:3] - Returns the first 2 characters of s
"friend".endswith("end") - Returns "friend"
"end" in s - Returns True if the substring "end" is found in s
```

---

**RANGE**

Range objects are useful for creating sequences of integers for looping.

```
range(5) - Returns a sequence from 0 to 4
range(2000, 2020) - Returns a sequence from 2000 to 2019
range(0,11,2) - Returns a sequence from 0 to 10, with each items incrementing by 2
range(0,-10,-5) - Returns a sequence from 0 to -9
list(range(2)) - Returns a list from 0 to 4
```

---

**DICTIONARIES**

```
max(d, key=key) - Returns the key that corresponds to the largest value in d
min(d, key=key, get=None) - Returns the key that corresponds to the smallest value in d
```

---

**SETS**

```
my_set = set([1]) - Returns a set object containing the unique values from 1
```

---

**DATE/TIME**

```
import datetime as dt - Import the datetime module
now = dt.datetime.now() - Assigns datetime.datetime object representing the current time to now
wck4 = dt.timedelta(hours=4) - Assign timedelta object representing a timespan of 4 hours to wck4
wck4 + dt.datetime.now() - Adds the current time to wck4
```

---

**def calculate(addition\_one, addition\_two, exponent\_val, factor=1,
 exponent\_val, factor=1):
 result = (value\_one+value\_two)\*\*exponent\*factor
 return result**

**LIST COMPREHENSION**

```
[i**2 for i in range(10)] - Returns a list of the squares from 0 to 9
[dict1 for s in l, string] - Returns the list of strings, each one having had the .lower() method applied
[i for i in l, float if i < 0.5] - Returns the items from l, floats that are less than 0.5
```

---

**FUNCTIONS FOR LOOPING**

```
for i, value in enumerate(l):
 print("The value of item {} is {}".format(i, value))
 - Iterates over the list l, printing the index location of each item and its value
for one, two in zip(one, two):
 print("{}: {}, {}: {}".format(one, two))
 - Iterates over two lists, one, one and two, and prints each item from both lists
 while x < 50:
 x += 1
 - Run the code in the body of the loop until the value of x is no longer less than 10
```

---

**TRY/EXCEPT**

```
try:
 print("I am about to divide by zero")
 1/0
except:
 print("I am about to divide by zero")
 - Catch and handle errors
 1/0
 - Convert each value of 1/int to a float, catching and handling ValueError: could not convert string to float: when values are missing.
```

This intermediate-level cheat sheet is a follow-up of the other Dataquest cheat sheet. It contains intermediate dtype methods, looping and handling errors.

## (9) Dataquest NumPy



This image shows a comprehensive cheat sheet for NumPy, organized into several sections:

- KEY:** We'll use shorthand in this cheat sheet  
arr - A numpy Array object
- IMPORTS:** Import these to start  
`import numpy as np`
- IMPORTING/EXPORTING:**
  - `np.loadtxt('file.txt')` - From a text file
  - `np.genfromtxt('file.csv', delimiter=',')` - From a CSV file
  - `np.savetxt('file.txt', arr, delimiter=' ')` - Writes to file the arr
  - `np.savetxt('file.csv', arr, delimiter=',')` - Writes to a CSV file
- CREATING ARRAYS:**
  - `np.array([(1,2,3)])` - One dimensional array
  - `np.array([(1,2,3),(4,5,6)])` - Two dimensional array
  - `np.zeros(3)` - 1D array of length 3 with all values 0
  - `np.ones(3)` - 1D array with all values 1
  - `np.random(3)` - 1D array of 3 random floats between 0 and 1
  - `np.linspace(0,100,6)` - Array of 6 evenly divided values from 0 to 100
  - `np.arange(0,10,3)` - Array of values from 0 to less than 10, with step size of 3 (`[0,3,6,9]`)
  - `np.full((2,3),2)` - 2x3 array with all values 2
  - `np.random(4,5)` - 4x5 array of random floats between 0.1 and 1.0
  - `np.random.rand(6,7)*100` - 6x7 array of random floats between 0 and 100
  - `np.random.randint(5,size=(2,3))` - 2x3 array with random ints between 0-4
- INSPECTING PROPERTIES:**
  - `arr.size` - Returns number of elements in arr
  - `arr.shape` - Returns dimensions of arr (rows, columns)
  - `arr.dtype` - Returns type of elements in arr
  - `arr.astype(dtype)` - Convert arr elements to type dtype
  - `arr.tobytes()` - Convert arr to a Python list
  - `np.info(np.array)` - View documentation for np.array
- COPYING/EDITING/RESHAPING:**
  - `np.copy(arr)` - Copies arr to new memory
  - `arr.view(dtype)` - Creates view of arr elements with type dtype
  - `arr.sort()` - Sorts arr
  - `arr.sort(axis=0)` - Sorts specific axis of arr
  - `two_d_arr.flatten()` - Flattens 2D array
  - `two_d_arr[10]` - Returns element at index 10
- TRANSPOSE:**
  - `arr.T` - Transposes arr (rows become columns and vice versa)
  - `arr.reshape(3,4)` - Reshapes arr to 3 rows, 4 columns without changing data
  - `arr.resize(5,6)` - Changes arr shape to 5x6 and fills new values with 0
- ADDING/REMOVING ELEMENTS:**
  - `arr.append(values)` - Appends values to end of arr
  - `arr.insert(0,values)` - Inserts values into arr before index 0
  - `arr.delete(0, axis=0)` - Deletes row on index 0 of arr
  - `arr.delete(4, axis=1)` - Deletes column on index 4 of arr
- COMBINING/SPLITTING:**
  - `np.concatenate([arr1,arr2],axis=0)` - Adds arr2 after arr1 along axis 0
  - `np.concatenate([arr1,arr2],axis=1)` - Adds arr2 as columns to end of arr1
  - `arr2[::2]` - Splits arr into 3 sub-arrays
  - `arr2[::3]` - Splits arr horizontally on the 0th index
- INDEXING/SLICING/SUBSETTING:**
  - `arr[5]` - Returns the element at index 5
  - `arr[2,3]` - Returns the 2D array element on index [2][3]
  - `arr[1:4]` - Returns array element on index 1 the value 4
  - `arr[1,3:10]` - Returns array element on index [1][3:10] the value 10
  - `arr[0:3]` - Returns the elements at indices 0,1,2 (On a 2D array: returns rows 0,1,2)
  - `arr[0:3,1]` - Returns the elements on rows 0,1,2 at column 1
  - `arr[::2]` - Returns the elements at indices 0,1,3 (On a 2D array: returns rows 0,1,3)
  - `arr[::1]` - Returns the elements at index 1 on all rows
  - `arr[:5]` - Returns an array with boolean values (`(arr>1)`)
  - `arr[::3] & (arr>2)` - Returns an array with boolean values
  - `~arr` - Inverts a boolean array
  - `arr[arr<5]` - Returns array elements smaller than 5
- SCALAR MATH:**
  - `np.add(arr,1)` - Add 1 to each array element
  - `np.subtract(arr,2)` - Subtract 2 from each array element
  - `np.multiply(arr,3)` - Multiply each array element by 3
  - `np.divide(arr,A)` - Divide each array element by A (returns np.nan for division by zero)
  - `np.power(arr,5)` - Raise each array element to the 5th power
- VECTOR MATH:**
  - `np.add(arr1,arr2)` - Elementwise add arr2 to arr1
  - `np.subtract(arr1,arr2)` - Elementwise subtract arr2 from arr1
  - `np.multiply(arr1,arr2)` - Elementwise multiply arr by arr2
  - `np.divide(arr1,arr2)` - Elementwise divide arr1 by arr2
  - `np.power(arr1,arr2)` - Elementwise raise arr1 raised to the power of arr2
  - `np.array_equal(arr1,arr2)` - Returns True if the arrays have the same elements and shape
  - `np.sqrt(arr)` - Square root of each element in the array
  - `np.sin(arr)` - Sine of each element in the array
  - `np.log(arr)` - Natural log of each element in the array
  - `np.abs(arr)` - Absolute value of each element in arr
  - `np.ceil(arr)` - Rounds up to the nearest int
  - `np.floor(arr)` - Rounds down to the nearest int
  - `np.round(arr)` - Rounds to the nearest int
- STATISTICS:**
  - `np.mean(arr, axis=0)` - Returns mean along specific axis
  - `arr.sum()` - Sum of arr
  - `arr.min()` - Returns minimum value of arr
  - `arr.max()` - Returns maximum value of arr
  - `np.var(arr)` - Returns the variance of array
  - `np.std(arr, axis=1)` - Returns the standard deviation of specific axis
  - `arr.corrcoef()` - Returns correlation coefficient of array

NumPy is at the heart of data science. Advanced libraries like scikit-learn, Tensorflow, Pandas, and Matplotlib all built on NumPy arrays. You need to understand NumPy before you can thrive in data science and machine learning. The topics of this cheat sheet are creating arrays, combining arrays, scalar math, vector math and statistics.

This is only one great NumPy cheat sheet—if you want to get more, check out our article on the 10 best NumPy cheat sheets!

## (10) Python For Data Science (Bokeh)

Want to master the visualization library [Bokeh](#)? This cheat sheet is for you! It contains all the basic Bokeh commands to get your beautiful visualizations going fast!

(11) Pandas Cheat Sheet for Data Science

Pandas is everywhere. If you want to master "*the Excel library for Python coders*", why not start with this cheat sheet? It'll get you started fast and introduces the most important Pandas functions to you.

You can find a best-of article about the 7 best Pandas Cheat Sheets here

(12) Regular Expressions Cheat Sheet

## Data Science Cheat Sheet Python Regular Expressions

**SPECIAL CHARACTERS**

- | Matches the expression to its right at the start of a string, it matches every such sentence before each ‘\n’ in the string.
- | Matching the expression to its left at the start of a string, it matches every such sentence after each ‘\n’ in the string.
- | Matches any character except line separator characters.
- | Escapes special characters or denotes character classes.
- | Any character from ‘a’ to ‘z’ or ‘A’ to ‘a’ matched first, it is left unshifted.
- | Greedy matches the expression to its left 1 or more times.
- | Greedy matches the expression to its left 0 or more times.
- | Greedy matches the expression to its left 0 or more times, if the expression is followed by ‘?’, it is added to qualify ‘\*’ or ‘+’.
- | Non-greedy matches the expression to its left 0 or more times.
- | Non-greedy matches the expression to its left 0 or more times, and not less than 1.
- | Match the expression to its left 0 or 1 times, and not less than 1.
- | Match the expression to its left 0 or 1 times, and ignores ‘?’. See also ‘?|’.

**SETS**

- | { } Contains a set of characters to match.
- | [ ]| Matches either ‘a’, ‘b’, or ‘c’, it does not include anything else in the set.
- | [x-z]| Matches any alphabet from ‘x’ to ‘z’.
- | [a-zA-Z]| Matches any character from ‘a’ to ‘z’ or ‘A’ to ‘Z’.
- | [^a-zA-Z]| Matches any character that is not ‘a’ to ‘z’ or ‘A’ to ‘Z’.
- | [^a-zA-Z ]| Matches any characters from ‘a’ to ‘z’ or ‘A’ to ‘Z’ and a space.
- | [^a-zA-Z ]+| Special characters become ‘bold’ when enclosed in brackets.
- | [^a-zA-Z ]\*| Adds ‘\*’ to the expression in the set. Here, it matches characters that are not in the set.

**CHARACTER CLASSES**

[A..Z, \w, \W, \d, \D]

- | \w| Matches word characters, which means ‘a’-‘z’, ‘A’-‘Z’, and ‘\_’.
- | \W| Matches non-word characters.
- | \d| Matches any digit, which means ‘0’-‘9’.
- | \D| Matches any non-digit.
- | \b| Matches the boundary of a word, that is, between words.
- | \B| Matches where ‘\w’ does not, that is, the boundary of ‘\w’ characters.

**POSIX REGULAR EXPRESSIONS**

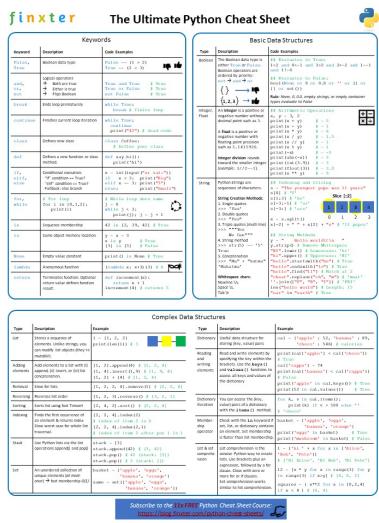
- | [ ]| Matches the expression inside the parentheses and groups it.
- | ( )| Inside parentheses like this, ‘x’ can be an expression.
- | [ ]\*| Matches the expression to its right 0 or more times.
- | [ ]+| Matches the expression to its right 1 or more times.
- | [ ]?| Matches the first instance of an expression ‘a’ in a string, it returns the index of the first occurrence.
- | re.match(‘a’, ‘abc’) | Match the expression matched by an ‘a’ in a group of ‘abc’.
- | (.+)| The number 1 corresponds to the first group of parentheses.
- | (.+g)| If there is a ‘g’ in the string, it matches more instances of the same group. If there is no ‘g’, it matches all of writing out the whole expression again. We can use ‘f’ or ‘g’ up to 99 such groups and their corresponding numbers.

**REGULAR EXPRESSION FUNCTIONS**

- | re.compile(‘a’) | Compiles the regular expression module.
- | re.findall(‘a’, ‘abc’) | Find all instances of ‘a’ in ‘abc’.
- | re.match(‘a’, ‘abc’) | Match the first instance of ‘a’ in ‘abc’.
- | re.search(‘a’, ‘abc’) | Search for the first occurrence of ‘a’ in ‘abc’.
- | re.sub(‘a’, ‘b’, ‘abc’) | Replace ‘a’ with ‘b’ in the string ‘abc’.

Regex to the rescue! **Regular expressions** are wildly important for anyone who handles large amounts of text programmatically (ask Google). This cheat sheet introduces the most important Regex commands for quick reference. Download & master these regular expressions!

## (13) The World's Most Concise Python Cheat Sheet



This cheat sheet is the most concise Python cheat sheet in the world. It contains keywords, basic data structures, and complex data structures—all in a single 1-page PDF file. If you're lazy, this cheat sheet is a must!

If you love cheat sheets, here are some interesting references for you (lots of more PDF downloads):



<https://gist.github.com/bgoonz/e4b30e43d5f1b8c79e9c6529d1cd0322>