datastructures-in-python

Home

data structures https://codepen.io/bgoonz/full/ZEyXNVr

This Website:

Home https://py-v2.gitbook.io/datastructures-in-pytho/

https://bgoonz42.gitbook.io/datastructures-in-pytho/datastructures-in-pytho/ https://bgoonz42.gitbook.io/datastructures-in-pytho/datastructures-in-pytho/

The Git Repo For This Website:

GitHub - bgoonz/python-gitbook https://github.com/bgoonz/python-gitbook

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

Blog:

Web-Dev-Hub
https://bgoonz-blog.netlify.app/

Web-Dev-Hub
https://master--bgoonz-blog.netlify.app/

Downloads:

https://bgoonz42.gitbook.io/datastructures-in-pytho/index

https://bgoonz42.gitbook.io/datastructures-in-pytho/index

For Beginners:

https://bgoonz42.gitbook.io/datastructures-in-pytho/misc/untitled

https://bgoonz42.gitbook.io/datastructures-in-pytho/misc/untitled

Navigation

Website Version

Table of contents

30 days of python practice

Understanding variables

Variables are simply declarations that are used to store certain values. For instance, the variable `name` can hold the value of `John Smith.` Several rules need to be considered when declaring variable names. For starters, a variable name cannot begin with a number. `2name = incorrect #incorrect` `name = correct #correct` Variable names are case sensitive. This means that the variable `school` is not the same as `School`. Variables can hold different data types. This includes strings, integers, Booleans, long, lists, and arrays. In Python, we do not need to declare the data type while writing a variable. This is because the code is compiled and interpreted later. The compiler will throw an error in case there is a mismatch in the data types. Let's talk about the different data types. 1. Strings Strings are usually presented in a text format. We will declare a string variable, as shown below.

```
name = “john”school = “Alliance Francaise”
```

When we run `print(name)`, the output will be `john`. 1. Integers These variables hold numeric values, as shown below.

```
math = 90chemistry = 100biology = 70
```

We can find the total of the variables above using the following statement.

```
print(math+chemistry+biology)
```

The total is `260` . A TypeError is thrown when you try to add a string to an integer, as shwon below.

```
var1 = "30" #stringvar2 = 20 #integerprint(var1+var2)#type error
```

We can sum `var1` and `var2` by converting `var1` to an integer using the `int()` function. The following code will execute successfully.

```
var1 = "30" #stringvar2 = 20 #integerprint(int(var1)+var2) # Output: 50
```

> Make sure that the variable stores a value that can be converted to an integer before using the int() method. 1. Booleans There are only two Boolean values: `True` and `False` . In other words, something can either be true or false. We declare these values, as shown below. Please note that Python is case sensitive.
>
> ```
> isOn = TrueisChecked = False
> ```
>
> A `bool()` method can help convert a value to a boolean. The code snippets below showcase how a `bool()` function can be used.

```
print(bool("abc")) #returns Trueprint(bool(0))  #returns False
```

The `bool()` function returns False when there are no parameters. 1. Float This data type consists of numbers that have a decimal place. A perfect example of a float variable is highlighted below.

```
Bmi = 45.7
```

Understanding lists

Lists allow us to store numerous elements in a particular variable. For instance, we can have a list that stores all the student names in a class. We use `[]` to define a list.

```
students = [] #list example
```

Elements in a list are usually separated by a comma, as shown below.

```
students = [“john”, “Mary Thomas”, “John Smith”]
```

Each element in the above `students` list has an index. By default, the first index is 0. So the item at index [0] is `john`, while the value at index `1` is `Mary Thomas`. A list of integers will look as follows.

```
student_marks = [90, 78, 90, 78]
```

We can access different list functionalities using built-in functions. For instance, to add a value to the `student_marks` list, we use the `append` function.

```
student_marks.append("Guardian Angel")print(student_marks)
```

The above function adds `Guardian Angel` at the end of the `student_marks` list. When we print the list it shows:

```
#output[90, 78, 90, 78, 'Guardian Angel']
```

We use `len(student_marks)` to determine the length of the list. We use the `remove()` function to delete something from the list. For instance, we can remove `90` from the `student_mark` list as shown below.

```
student_marks.remove(90)print(student_marks)
```

In lists, negative indices allow us to count elements starting from the last one. For instance, the element with an index of `-1` in the above `student_marks` list is `"Guardian Angel"`. The second last element `78` has an index of `-2`.

Understanding functions or methods

Methods are quite critical in programming. They help store reusable code. This means that a person can call already declared methods rather than writing statements from scratch repeatedly. This saves significant time, that can be invested in other productive activities. In Python, we use the `def` keyword to declare a function. An example of a python method is shown below.

```
def readData():    print('success')
```

The above function prints `success` when it's invoked. We can also pass data to a method, perform some calculations, and return the results. This is demonstrated in the code snippet below.

```
def calculateTotal(chem, bio):    return chem+bioprint(calculateTotal(90,8
```

The `calculateTotal` method takes in two parameters (chem, bio). The function then returns the sum of the two values. It is important to take note of the data types when passing parameters. For instance, the `calculateTotal` method will not work when we pass in a string as a parameter. This is because the program cannot sum up an integer and a string. As shown above, we can call the `calculateTotal` method directly from our print statement.

```
print(calculateTotal(90,80))
```

The `return` keyword ensures that the method returns a result after execution. Note that a function can also call another method. This is illustrated below.

```
def readData(chem, bio):    return chem+biodef getTotal():    print(readDa
```

Understanding loops

Loops are critical because they allow us to iterate through lists, check for different conditions, and continuously execute various statements. The main loops are `for` and `while`. 1. For loops As noted, we can use a for loop to iterate through a list, as shown below:

```
student_list = [“John Doore”,”Matu Smith”]for x in student_list:    print(
```

The `for` loop above will print every item in the student_list. 1. While loops A while loop can help us check for a particular condition. For instance, while something is true specific statements can be executed. Here is an example of a while loop in action.

```
isChecked = falsewhile isChecked == true:    print('Hallo there')
```

Note that the while loop above will be executed indefinitely until isChecked is set to false. You can press ctrl+c to stop the loop.

Classes

Classes are a vital component of object-oriented programming. When creating a class, you must use the `class` keyword. Other elements are then nested in the class. Here is an example of a Python class.

```
class Farmer: # a class with the name farmer    name = "John" # A variable
```

Classes can help as group things with similar characteristics. We can also assign values to class variables using the `init` function.

```
class Farmer:  def __init__(self, farmername, produce):    self.farmername
```

In the above `Farmer` class, the `self` keyword represents an instance of an object. In other words, it allows us to access the different methods and attributes defined in the class. You can also declare a method in a class and use it later, as shown below.

```
class Farmer:  def __init__(self, farmername, produce):    self.farmername
```

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

Lets create a python 3 repl, and call it *Hello World*. Now you have a blank file called *main.py*. Now let us write our first line of code: *helloworld.py*

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

Brian Kernighan actually wrote the first “Hello, World!” program as part of the documentation for the BCPL programming language developed by Martin Richards. Now, press the run button, which obviously runs the code. If you are not using replit, this will not work. You should research how to run a file with your text editor. If you look to your left at the console where hello world was just printed, you can see a `>`, `>>>`, or `$` depending on what you are using. After the prompt, try typing a line of code.

```
Python 3.6.1 (default, Jun 21 2017, 18:48:35)[GCC 4.9.2] on linuxType "hel
```

The command line allows you to execute single lines of code at a time. It is often used when trying out a new function or method in the language. Another cool thing that you can generally do with all languages, are comments. In python, a comment starts with a `#`. The computer ignores all text starting after the `#`. *shortcom.py*

```
# Write some comments!
```

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

```
'''Dear PYer,I am confused about how you said you could use triple quotes
```

Unlike many other languages, there is no `var`, `let`, or `const` to declare a variable in python. You simply go `name = 'value'`. *vars1.py*

```
x = 5y = 7z = x*y # 35print(z) # => 35
```

Remember, there is a difference between integers and strings. *Remember: String =* `""`. To convert between these two, you can put an int in a `str()` function, and a string in a `int()` function. There is also a less used one, called a float. Mainly, these are integers with decimals. Change them using the `float()` command. *vars2.py*

```
x = 5x = str(x)b = '5'b = int(b)print('x = ', x, '; b = ', str(b), ';') #
```

Instead of using the `,` in the print function, you can put a `+` to combine the variables and string. There are many operators in python:

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

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

simpleops.py

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

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

complexop.py

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

The ones above are to edit the current value of the variable. Sorry to JS users, as there is no `i++;` or anything.

Fun Fact: The python language was named after Monty Python.

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

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

quotes.py

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

slicing.py

String Slicing

You can look at only certain parts of the string by slicing it, using `[num:num]`. The first number stands for how far in you go from the front, and the second stands for how far in you go from the back.

```
x = 'Hello everybody!'x[1] # 'e'x[-1] # '!'x[5] # ' 'x[1:] # 'ello eve
```

Methods and Functions

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

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

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

strings.py

```
x = " Testing, testing, testing, testing       "print(x.strip())print(
```

Good luck, see you when you come back!

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

inp.py

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

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

inp2.py

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

Running: *inp.py*

```
Type something:Hello WorldHere is what you said: Hello World
```

inp2.py

```
Type something: Hello WorldHere is what you said: Hello World
```

Python has created a lot of functions that are located in other .py files. You need to import these **modules** to gain access to the,, You may wonder why python did this. The purpose of separate modules is to make python faster. Instead of storing millions and millions of functions, , it only needs a few basic ones. To import a module, you must write `input <modulename>`. Do not add the .py extension to the file name. In this example , we will be using a python created module named random.

module.py

```
import random
```

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

module2.py

```
import randomprint(random.randint(3,5)) # Prints a random number betwe
```

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

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

for Loop

The for loop goes through a list of variables, making a seperate variable equal one of the list every time. Let's say we wanted to create the example above.

loop.py

```
from time import sleepfor i in range(100):     print('Hello')     slee
```

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

loop2.py

```
import timefor number in range(100):     print(number)     time.sleep(
```

while Loop

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

while.py

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

Or you could do:

while2.py

```
import randomposition = '<placeholder>'while position != 1: # will run
```

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

if.py

```
import randomnum = random.randint(1, 10)if num == 3:     print('num is
```

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

elif.py

```
import randomnum = random.randint(1, 10)if num == 3:    print('Num is
```

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

else.py

```
import randomnum = random.randint(1, 10)if num == 3:    print('Num is
```

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

Characters: 389

nofunc.py

```
import randomprint(random.randint(1, 9))print('Wow that was interestin
```

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

Characters: 254

functions.py

```
import randomdef r(t):    print(random.randint(1, 9))    print(t)r('
```

Chapter 01 - Getting Ready with Python

Installing Python 3, And Launching Python Shell

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

- Installing Python 3 and Launch Python Shell

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

A quick word of caution on Windows

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

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

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

Command to launch Python 3 is different in Mac.

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

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

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

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

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

Chapter 02 - Introduction To Python Programming

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

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

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

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

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

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

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

Summary

In this step, we:

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

Step 01: Our First Programming Challenge

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

The Print Multiplication Table Challenge (PMT-Challenge)

- Compute the multiplication table for `5`, with entries from `1` to `10`.

- Display this table.

  The display needs to be:

  5 * 1 = 5

  5 * 2 = 10

  5 * 3 = 15

  5 * 4 = 20

  5 * 5 = 25

  5 * 6 = 30

  5 * 7 = 35

  5 * 8 = 40

  5 * 9 = 45

  5 * 10 = 50

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

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

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

Summary

In this step, we:

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

Step 02: Breaking Down PMT-Challenge

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

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

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

Here is how our draft steps look like

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

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

Summary

In this step, we:

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

Step 03: Introducing Operators And Expressions

Let's focus on solving the first sub-problem of *PMT-Challenge*, the numeric computation. We want the computer to calculate `5 * 5` for example, and print `25` for us. How do we get it to do that? That's what we would be looking at in this step.

Snippet-01: Introducing Operators

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

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

```
>> 5 X 5    File "< stdin >", line 1    5 X 5       ^    SyntaxError: i
```

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

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

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

"*5 into 5*" is achieved by the code `5 * 5`, and you can see the result `25` being printed. Similarly, `5 * 6` gives us `30`.

```
>> 5 * 6    30
```

There are a wide range of other operators in Python:

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

```
>> 5 + 611>>> 5 - 6-1
```

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

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

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

Let's look at some terminology:

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

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

```
>> 5 + 5 + 5    15    >>> 5 + 5 * 5    30
```

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

Summary

In this step, we:

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

Step 04: Programming Exercise IN-PE-01

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

Exercises

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

Note

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

Solutions

Solution 1

```
>> 24 * 60    1440
```

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

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

Solution 2

```
>> 24 * 60 * 60    86400
```

How many seconds are there in a day?

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

Summary

In this step, we:

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

Step 05: Puzzles On Expressions

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

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

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

Snippet-01: Puzzles On Expressions

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

```
>> 5 $ 2    File "< stdin >", line 1    5 $ 2    ^    SyntaxError:
```

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

```
>> 5+6+10    21
```

In an expression, using spaces makes it easier for you to read it, but it’s not mandatory. `5 + 6 + 10` is easier to read than `5+6+10`, but does not make any difference to the Python compiler.

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

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

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

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

```
>> 5 + 5 * 6    35
```

The correct result is `35`.

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

Operators in Python are divided into two sets as follows:

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

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

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

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

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

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

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

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

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

Summary

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

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

Step 06: Printing Text

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

Snippet-01: Printing Text

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

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

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

```
>> Hello    Traceback (most recent call last):      File "<stdin>", li
```

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

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

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

```
>> print Hello      File "<stdin>", line 1        print Hello
```

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

Will `print(Hello)` work?

```
>> print (Hello)    Traceback (most recent call last):      File "<std
```

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

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

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

```
>> print ("Hello")    Hello    >>> print("Hello")    Hello
```

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

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

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

What exactly are we trying to print?

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

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

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

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

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

Summary

In this step, we:

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

Step 07: Puzzles On Utility Methods, And Strings

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

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

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

```
>> print("5*6")    5*6    >>> print('5*6')    5*6
```

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

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

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

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

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

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

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

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

```
>> abs 10.5      File "<stdin>", line 1        abs 10.5
```

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

```
>> abs("10.5")    Traceback (most recent call last):      File "<stdin
```

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

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

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

You'll see that `32` is printed.

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

```
>> pow(10, 3)    1000    >>> 10 ** 3    1000
```

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

```
>> max(34, 45, 67)    67    >>> min(34, 45, 67)    34
```

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

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

```
>> pow(2,5)     32     >>> Pow(2,5)     Traceback (most recent call last)
```

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

```
>> print("Hello")     Hello     >>> print("hello")     hello     >>> print
```

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

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

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

```
>> print ( "hellO World" )     hellO World     >>> print ( "hellO     Wo
```

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

```
>> print("Hello"")      File "<stdin>", line 1        print("Hello"")
```

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

```
>> print("Hello\"")Hello">>>
```

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

```
>> print("Hello\nWorld")    Hello    World
```

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

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

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

```
>> print("Hello\\World")     Hello\World     >>> print("Hello\\\\\\World
```

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

```
>> print('Hello"')     Hello"     >>> print("Hello'World")     Hello'Worl
```

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

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

Summary

In this step, we:

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

Step 08: Formatted Output With print()

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

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

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

Snippet-01: print() Formatted Output

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

Let's see an example:

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

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

How do we get `10` to be printed then?

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

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

`"VALUE {0}"` is what we need.

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

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

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

```
>> print("VALUE {0}".format(10,20,30))    VALUE 10    >>> print("V
```

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

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

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

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

Cool! Progress made.

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

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

The great thing about this, is now we can replace the values we passed to `print()` in the first place, without changing the indexes! So, we can display results for `5 * 7 = 35` and `5 * 8 = 40`. We are now able to print `5 * 6 = 30`, `5 * 7 = 35`, `5 * 8 = 40`, and can do similar things for other table entries as well.

```
>> print("{0} * {1} = {2}".format(5,7,5*7))    5 * 7 = 35    >>> p
```

Summary

In this step, we:

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

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

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

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

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

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

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

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

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

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

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

```
>> print("{0} * {1} = {4}".format(5,8))    Traceback (most recent call
```

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

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

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

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

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

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

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

Summary

In this step, we:

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

  Step 10: Introducing Variables

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

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

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

  Let's try a different approach.

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

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

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

```
>> index = 2
```

Aha! This compiles.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Snippet-02: Classroom Exercise On Variables

Let's do a simple exercise with variables.

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

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

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

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

```
>> a = 5    >>> b = 6    >>> c = 7    >>> print("5 + 6 + 7 = 18")
```

The way you can do that is with code like this: `print("{0} + {1} + {2} = {3}".format(a,b,c,a+b+c))`.

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

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

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

```
>> a = 6    >>> b = 7    >>> c = 8    >>> print("{0} + {1} + {2} =
```

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

Summary

In this step, we:

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

Step 11: Puzzles On Variables

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

We will start with looking at a few puzzles.

Snippet-01: Puzzles On Variables

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

```
>> count    Traceback (most recent call last):     File "<stdin>", li
```

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

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

```
>> count = 4    >>> print(count)    4
```

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

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

Python will create a variable in its memory.

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

```
>> Count    Traceback (most recent call last):      File "<stdin>", li
```

There are rules to follow while naming variables.

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

```
>> 1count = 5      File "<stdin>", line 1        1count = 5
```

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

```
>> c12345 = 5
```

To summarize the rules for naming variables.

- This should start with an alphabet (a capital or a small alphabet) or underscore.

- Starting the second character, it can be alphabet, or underscore, or a numeric value.

Summary

In this step, we:

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

Step 12: Introducing Assignment

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

Snippet-01: Introducing Assignment

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

```
>> i = 5    >>> j = i    >>> j    5
```

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

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

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

`j` refers to a value of `10`

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

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

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

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

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

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

```
>> num1 = 5    >>> num2 = 3    >>> sum = num1 + num2    >>> sum    8
```

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

```
>> a = 5    >>> b = 6    >>> c = 7    >>> sum = a + b + c    >>> sum
```

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

Summary

In this step, we:

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

Step 13: Introducing Formatted Printing

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

This is the approach we used until now.

```
>> a = 1    >>> b = 2    >>> c = 3    >>> sum = a + b + c    >>> print
```

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

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

```
>> print(f"")    >>> print(f"value of a is {a}")    value of a is 1
```

The variable within braces is replaced by its value.

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

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

This feature was introduced in a Python 3 release.

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

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

You can see how easy it turns out to be!

Step 14: The PMT-Challenge Revisited

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

Snippet-01:

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

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

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

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

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

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

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

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

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

```
>> for i in range(1,10):    ...    print(i)    ...    1    2    3    4
```

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

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

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

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

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

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

How can you do that?

Let's start with a simple example.

```
>> for i in range(1,11):    ...    print(f"{i}")    ...    1    2    3
```

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

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

```
>> for i in range(1,11):    ...    print(f"5 * {i} = {5 * i}")    ...
```

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

Step 15: Loops

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

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

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

Snippet-01:

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

```
>> for i in range(1,10)      File "<stdin>", line 1        for i in ra
```

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

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

```
>> for i in range(1,10):    ... print(i)      File "<stdin>", line 2
```

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

```
>> for i in range(1,10):    ...    print(i)    ...    1    2    3    4
```

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

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

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

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

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

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

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

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

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

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

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

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

Summary

In this step, we:

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

Step 16: Programming Exercise PE-BA-02

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

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

Exercises

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

Solution 1

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

```
>> for i in range (2,11,2):    ...    print(i)    ...    2    4    6
```

Solution 2

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

```
>> for i in range (10,0,-1):    ...    print(i)    ...    10    9    8
```

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

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

Solution 3

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

```
>> for i in range (20,0,-2):    ...    print(i)    ...    20    18    1
```

Solution 4

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

```
>> for i in range (1,11):    ...     print(i * i)    ...    1    4
```

Solution 5

Let’s print the squares in the reverse order.

```
>> for i in range (10,0,-1):    ...     print(i*i)    ...    100    81
```

Solution 6

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

```
>> for i in range (10,0,-2):    ...     print(i*i)    ...    100    64
```

The key part is using a step of `-2`

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

Summary

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

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

Step 17: Review: The Basics Of Python

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

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

- `1` , `11` , `5` , ... are all called literals because these are constant values. Their values don't really change. _Consider `` 5 _ 4 _ 50`. This is an expression. `_`is an operator, and`5`, `4`and`50 `` are operands.
- The name `i` in `i = 1` , is called a variable. It can refer to different values, at different points in time.
- `range()` and `print()` are in-built Python functions.
- Every complete line of code is called statement. The specific statement `print()` , is invoking a method. The other statement which we looked at earlier, was an

assignment statement. `index = index + 1` would evaluate `index + 1`, and have the `index` variable refer to that value.

- The syntax of the `for` loop was very simple. `for var in range(1, 10) : ...`, followed by statements you would want to execute in a loop, with indentation. For the sake of indentation we left four `<SPACE>`s in front of each statement inside the `for` loop.

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

Chapter 03 - Introducing Methods

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

Snippet-01: Current Solution To PMT-Challenge

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

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

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

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

```
>> print_multiplication_table(7)    Traceback (most recent call last):
```

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

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

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

Step 01: Defining Your First Method

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

Snippet-01:

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

```
>> print("Hello World")    Hello World    >>> print("Hello World")
```

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

The syntax to create a method in Python is straightforward:

- At the start, use the keyword `def` followed by a space.
- Followed by name of the method - `print_hello_world_twice` .

- Add a pair of parenthesis: `()`.
- This is followed by a colon `:` (similar to what we used in a `for` loop).

```
>> def print_hello_world_twice():...     print("Hello World")...     p
```

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

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

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

```
>> print_hello_world_twice    <function print_hello_world_twice at 0x1
```

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

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

```
>> print_hello_world_twice()    Hello World    Hello World    >>> prin
```

Now, we are able to run the method.

Summary

In this step, we:

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

Step 02: Programming Exercise PE-MD-01

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

Exercises

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

  Solutions

  Solution 1

```
>> def print_hello_world_thrice():    ...     print("Hello World")
```

  Solution 2

```
>> def print_your_progress():    ...     print("Statement 1")    .
```

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

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

```
>> def print_your_progress():    ...     print("Statement 1\nStatem
```

We are using the newline character `\n` .

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

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

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

`print_your_progress()` represents a method call. The code inside the method is executed.

Summary

In this step, we:

- Implemented solutions to a few exercises that test our understanding of Python methods. We touched concepts such as:

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

Step 03: Passing Parameters To Methods

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

Snippet-01:

```
>> print_hello_world_twice()    Hello World    Hello World    >>>
```

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

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

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

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

How do we define our method to accept this argument?

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

```
>> def print_hello_world(no_of_times):    ...    print("Hello Worl
```

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

```
>> print_hello_world()    Traceback (most recent call last):
```

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

```
>> print_hello_world(5)    Hello World    5    >>> print_hello_wor
```

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

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

Think about using something along the lines of a loop.

Snippet-02:

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

```
>> def print_hello_world(no_of_times):    ...     for i in range(1,
```

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

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

```
>> def print_hello_world(no_of_times):    ...     for i in range(1,
```

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

Why is it not printing `5` times?

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

```
>> def print_hello_world(no_of_times):    ...     for i in range(1,
```

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

```
>> print_hello_world(7)     Hello World     Hello World     Hello Wor
```

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

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

Summary

In this step, we:

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

Step 04: Classroom Exercise CE-MD-01

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

Exercises

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

Solutions

Solution 1

```
>> def print_numbers(n):    ...    for i in range(1, n+1):    ...
```

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

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

Solution 2

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

```
>> def print_squares_of_numbers(n):    ...    for i in range(1, n+1):
```

How is a parameter different from an argument?

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

Summary

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

Step 05: Methods With Multiple Parameters

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

Snippet-01:

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

```
>> def print_hello_world(no_of_times):    ...    for i in range(1,
```

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

You can always create another method similar to the first one, such as `print_welcome_to_python(no_of_times)` and print the necessary text inside.

However, is that what a good programmer does?

A good programmer tries to create a more generic solution.

```
>> def print_string(str, no_of_times):    ...    for i in range(1,
```

The good programmer that you are, you created a new method called `print_string(str, no_of_times)` accepting a text parameter, in addition to `no_of_times` .

Syntax rules for method parameters are quite strict. If we say `print_string("Welcome to Python")` and run it, we get an error! Python Shell says: “I need `no_of_times` to be present in here”.

```
>> print_string("Welcome to Python")    Traceback (most recent cal
```

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

The Python language makes this very easy. `def print_string(str = "Hello World", no_of_times=5)` . The rest of the method remains the same.

```
>> def print_string(str="Hello World", no_of_times=5):    ...    f
```

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

```
>> print_string()    Hello World    Hello World    Hello World
```

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

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

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

```
>> print_string("Welcome to Python", 8)    Welcome to Python    We
```

Isn’t that cool!

Summary

In this step, we:

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

Step 06: Back To Multiplication Table - Using Methods

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

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

Snippet-01:

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

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

```
>> def print_multiplication_table(table):    ...    for i in range(1,1
```

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

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

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

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

```
>> def print_multiplication_table(table, start, end):    ...    for i
```

Simple! Define those range limits as additional parameters!

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

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

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

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

Summary

In this step, we:

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

Step 07: Indentation Is King

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

Snippet-01:

```
>> def method_to_understand_indentation():    ...     for i in range(1
```

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

```
>> def method_to_understand_indentation():    ...     for i in range(1
```

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

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

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

```
>> def method_to_understand_indentation():    ...     for i in range(1
```

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

```
>> method_to_understand_indentation()    1    5    2    5    3    5
```

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

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

Summary

In this step, we:

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

Step 08: Puzzles on Methods - Named Parameters

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

Snippet-01:

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

```
>> def print_string(str="Hello World", no_of_times=5):    ...     for
```

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

```
>> print_string(6)    6    6    6    6    6
```

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

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

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

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

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

```
>> print_string(no_of_times=6)    Hello World    Hello World    Hello
```

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

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

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

```
>> print_string(7, 8)    7    7    7    7    7    7    7    7
```

You would see that `7` is printed `8` times.

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

```
>> print_string(7.5, 8)    7.5    7.5    7.5    7.5    7.5    7.5    7
```

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

```
>> print_string(7.5, "eight")    Traceback (most recent call last):
```

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

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

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

Snippet-02:

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

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

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

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

```
>> def 1_print():      File "<stdin>", line 1        def 1_print():
```

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

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

Methods and variables cannot be named using Python keywords.

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

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

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

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

```
>> def def():      File "<stdin>", line 1        def def():
```