Python 3 Cheat Sheets

Scientific Computing 118 Supplement

Sejin Kim, Kenyon College

The WTFPL License

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
Version 2, December 2004

Copyright (C) 2019 Sejin Kim

Everyone is permitted to copy and distribute verbatim or modified copies of this license document, and changing it is allowed as long as the name is changed.

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. You just DO WHAT THE FUCK YOU WANT TO.

Why study development?

Software is becoming a larger and larger part of our daily lives. Different machines use different forms of software. From ATMs to parking meters, compute servers to emergency radios, software runs our lives increasingly. In fact, if all of the software stopped working, planes would just drop out of the skies, stocks would crash instantly, and global communications would cease to exist.

Python is only one of many different programming languages out there, and it has its own special place in the development world. Other languages include C++, Haskell, Ruby, R, Java, JavaScript, and even Ada. Once you pick up Python, though, you'll find that other programming languages don't look so different.

Of note is, development, to most people, is not inherently fun. I know, right? In reality, the act of coding is full of twists and turns, and it's not easy. However, the joy of coding comes with the things that you can do with your code. Think of code as a tool that you can do so many other things with.

What is this text not? It's not a comprehensive view of the software development landscape, or the hardware or firmware world. We won't discuss what a CPU does or how it works, or how a temperature sensor on a motherboard sends input. This text will also not provide you with any development exercises. However, we encourage you to walk through coding exercises on your own. This is only an introductory text for reading and developing in Python.

0. Development Basics

Like a builder might use a hammer, a screwdriver, and a wrench, a programmer has some tools at their disposal. We use these tools to debug our code and turn it into instructions that our computer can understand.

The first and foremost is called an IDE, or integrated development environment. That's just a fancy way of saying a place where you can write and run code that you write. repl.it, Google Colaboratory, Apple Xcode and Microsoft Visual Studio are all IDEs. IDEs can be broken down into two fundamental components: a text editor and a compiler.

A text editor is pretty simple. It's a place where you can edit plain text. If you're on a Mac, you can use Apple Textedit, and if you're on PC, you can use Notepad. Don't confuse these with software packages like Microsoft Word, Google Sheets, or Apple Pages. You can also download different text editors, like BBEdit for Mac, Atom for Mac, PC and Linux, or Notepad++ for PC and Linux, and they all have their own strengths and weaknesses. There's also text editors that can run in the terminal, like emacs and nano.

A compiler is a little bit more complicated. We code in Python, but computers can't natively understand Python. Python would be what developers call "source code", and it's almost always human readable, but not machine readable. (There are some source code languages that aren't human readable, like BrainF, but these are fringe cases.) When we want to run a bit of our source code, we need to translate it into something that our computers can understand. This happens in two stages. The first stage converts our source code into assembly code, instructions that the computer can then break down. Assembly code is only readable to the trained eye, and is difficult to think about intuitively. The second stage breaks code down to machine code, or the 1's and 0's that the computer can understand, but that isn't human-readable.

All of that seems a little bit abstract still, so let's look at how that code looks in real life, if we were to open the files as a plain text file.

First, let's examine a sample Python script:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
print(factorial(5))
```

With a little bit of sleuthing, we can figure out that this script will find the factorial of 5, which turns out to be 120.

Now, let's look at the "human-readable" assembly code that our compiler will turn this into.

```
^C
<*^c^@^@^@^@^@^@^@^@^B^@^@^@@^@^@^@s^X^@^@^@d^@^@^^@^@Z^@^@e^@^@d^A^@^
^A^@GHd^B^@S(^C^@^@^@c^A^@^@^@^A^@^@^@^D^@^@^@$
```

Now that doesn't make sense. But, it makes perfect sense to our computer, which knows how to parse this information into an executable program.

One of the most important things to understand about computers is, they're not magic. It's not a black box. You can do as little or as much as you want with them, but they can do a lot for you. Have fun.

1. Printing and Variables

If there's anything that computers are good at doing, it's following instructions.

One of the most simple instructions that we can assign a program are prints. Printing is computationally virtually zero-effort, and it can enable a slew of debugging down the line. For example, take this Python program from above (don't worry, you don't have to understand what it does yet).

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
print(factorial(5))
```

All of that's pretty intimidating. Now, let's imagine that something's not working as we wanted it to. Something's getting hung up somewhere, but our program is still technically compiling. We could put some print statements in, perhaps more frequently, so that we could figure out exactly where things are going wrong.

Here's how that might look:

```
def factorial(n):
  if n==0:
    print("n is zero!")
    return 1
  else:
    return n * factorial(n-1)
    print("we can do the factorial!")
print(factorial(5))
```

As you can see, Python makes printing really easy. The print() function is a basic feature of base Python. No semicolons, no no chevrons, just `print()`.

We can also make things called variables. Like variables in math, program variables hold some data, but that data can change (hence the "vary" part). You could name a variable anything that isn't a reserved word in Python. You don't even have to use English. You could name a variable
' ¯_(ツ)_/¯ ' if you really wanted to. However, you probably want to name your variables well. They should describe what they hold.

There are several different types of variables, and they all have their advantages and disadvantages:

- `int` variables can only hold integers, or whole numbers; you cannot put letters or characters other than - and whole numbers into an integer
- `float` variables can hold decimals; in addition to the information that you could store in an int, you can now use a decimal point `.` to increase your precision. Now, you might be asking, why not just use a float for everything? Why do ints exist? Floats take dramatically more memory to hold. For a more concrete example check out the demo here[1], where you can see how the top line represents a big loop with an int, and the bottom line represents the same loop, but with a float.
- `complex` variables can also hold imaginary numbers; think back to Algebra 2. Remember how `sqrt(-1)` can be represented as an i in an equation? Complex variables can also hold imaginary numbers, and define them using a `j`. For example, 2*i in math would be written as `2j` in a complex variable.
- `string` variables can hold letters as information. Imagine that you had alphabet soup, and you picked out the "alphabet" part. If you got a piece of twine and put your alphabet on that twine in a particular order, you got yourself a Python string.
- `boolean` variables can only hold one information state: true or false. Benefits? They only require one bit to store, meaning that they're really fast to work with.

There are two steps to using variables in any programming language: **declaration** and **initialization**. Declaration just tells the compiler to make a new variable, and to give it the name that you've given it. You're *declaring* that this variable exists. Initialization is a fancy way of saying that we want to put something into a variable that we've declared. For example, we might declare a boolean variable called `varsityAthlete`, but it has no meaning until we initialize it with some data, like `True`.

Variables exist in almost every single modern programming language. Different programming languages deal with variables differently, but again, Python makes things pretty easy, since it doesn't require you to explicitly define the variable type. By using a single equal sign `=`, we can initialize and declare a variable in one fell swoop. Python will do its best to figure out what kind of variable it is, depending on what kind of information you give it. Let's look at an example.

```
cashValue = 3.39
accountHolder = "Stinky Pete"
isBroke = True
kitties = 390
```

[1] https://processing.org/examples/integersfloats.html The code shown is in the Processing programming language, a derivative of C++.

In our example, lines 1-3 are declaring three variables of differing types: a float, a string, and a bool. The variable names `cashValue`, `accountHolder`, and `isBroke` explain what information the variables hold pretty well. However, the variable `kitties` doesn't really make sense. Is this how many kitties I have? Is it my love or hatred of cats on a scale from 1-1000? We don't really know.

We talked about how Python is smart enough to figure out what kind of variable you're declaring, without having to explicitly declare the variable, like in some languages[2]. But, how does Python really know what you're trying to give it?

If you declare a variable with no quotes at all, but only with numerical information, Python will assume that you are declaring either an `int`eger or a `float`. By default, Python uses the smallest type possible; that is, if you declare a variable as 3, Python will try and store it as an `int`eger first. If you declare a variable as 3.39, Python knows that it can't store that as an `int`eger, so it'll upgrade the variable type to a `float`.

If you declare a variable with no quotes, but as a `True` or `False`, Python recognizes these reserved words as values of a `bool`ean variable. If the variable *only* stores a `True` or `False`, then Python will continue to store this value as such. You cannot attempt to put some other value into a `bool`ean variable without risking some unexpected behavior.

If you declare a variable with quotation marks or single quotes, Python will automatically assume that you are inputting a `string`. Whether the input is `"sandwiches"` or `"3"`, Python will still store that value as a `string`, and it is inoperable by standard arithmetic operations. Take a look at this code.

```
a = 3
b = "3"
c = a + b
print(c)
```

Intuitively, to humans, you'd think that this would print a 6. If you try to run this though, you'll probably get a concatenation error. That means that Python can't combine the types of variables into one.

Speaking of which, can we combine variables? Yes, yes we can. Remember how Python tries to store variables in the smallest type possible? Python can also "upgrade" variables of the same group of variables - that is, small numbers can become bigger. Take a look at this code.

[2] For example, in C++, when you declare a variable, you must declare its type as well: `float cashValue = 3.39;`

```
a = 3
b = 5.34
a = b + a
print(a)
```

The variable `a` is originally being declared as an integer variable, and `b` is being declared as a float variable. However, when I re-initialize `a` using a variable that's bigger than an integer, Python will decide that the variable can no longer be stored as an integer, and will upgrade the type to a float to accommodate the decimals. In other words, Python tries not to throw away information, if it can help it.

As alluded to above, we can also re-initialize variables, even with themselves. When we do so, the order matters! The data that will fill the variable should always come on the right, and the result should always be on the left. As such, the result can only be a single variable.

As a result, this is valid syntax.

```
a = b + c
```

But, this is not.

```
b + c = a
```

Variables are the foundational building block of any modern programming language. Whether you're talking about Python, Java, or C++, variables form the base that we'll build upon later.

2. Non Primitive Data Types

Python can also handle something called non-primitive data types. First, though, we need to define what a primitive data type is.

A primitive data type is some data variable that only stores one thing. That thing might be an integer, a boolean, or even a string, but critically speaking, it only stores one of those things.

Comparatively speaking, a non-primitive data type stores more than one of those things per type declaration. One of the most stimple types of declaration is a list. A list is simply a bunch of a single variable type, like a list of strings or a list of integers.

We can declare and initialize a list by enclosing the data in square brackets. Consider the following code.

```
primes = [2, 3, 5, 7]
```

In this, we're declaring a new list of integers named `primes`, and filling it the the data in the square brackets, a bunch of integers. Note that you cannot mix primitive variable types in one string. For example, consider the following code.

```
puppy = ["doberman", 5, 3.22]
```

This is not a valid declaration, syntactically. It mixes strings, integers, and floats, all in one list. Instead, we should declare these values separately. Consider the following code.

```
puppyBreeds = ["doberman", "pug", "golden lab"]
puppyWeights = [5, 3, 3]
puppyTemperments = [3.22, 4, 5.5]
```

In this example, `puppyBreeds` is a list of strings, `puppyWeights` is a list of integers, and `puppyTemperments` is a list of floats. Note that in the `puppyTemperments` list, there's an integer. However, because the list has been declared as a list of floats, our integer will be stored as a float.

3. Handling Numbers

Now that we know how to deal with variables, we can start manipulating numbers by putting them in variables. Our ability to manipulate numbers will allow us to control our programs and do more with less code.

There are seven fundamental number operations in Python:

```
(1+1) #addition
(1-1) #subtraction
(3*3) #multiplication
(3**3) #exponentiation
(6/2) #division
(7//3) #strict division
(28%3) #modulus
```

Addition, subtraction, multiplication, and division are (hopefully) pretty obvious, but exponentiation, strict division, and the modulus operators might not be[3].

The exponentiation operator uses two asterisks `**`, which tells the compiler to use the first number as the base, and the second number as the exponent. For example, if we were to write 2^9 on paper, we would write `(2**9)` in Python.

Strict division will drop any of the decimal places, making your answer correct to the ones place. If we ran `(9/2)`, we would get 4.5, but if we ran `(9//2)`, we would only get 4. When running strict division, Python will *never* round up. Even if we ran `(1399//100)`, Python will still return 13, not 14.

The modulus is closely related to division, and it's the computer science term for the remainder of a division operation. For example, if we ran `(9/2)`, we would get 4.5, but if we ran `(9%2)`, we would get 0.5.

It might be easiest to think about these operators in a table, especially when you're trying to get certain numbers out of a longer integer or float. An abridged example is given here.

1397			
	/	//	%

[3] Some documentation writes this as a modulo operator, while others write it as a modulus operator. They are the same thing.

1	1397	1397	1397
10	139.7	139	7
100	13.97	13	97
1000	1.397	1	397

When running arithmetic operations in Python, more parentheses never hurt. You should always enclose your basic operations in parentheses, and control the order of operations more tightly by dictating what operation to carry out first.

As specified in section 1, Python tries not to throw data away. If you specify an integer, and try to upgrade it to a float, Python should honor that. Likewise, if a float's decimal places are removed, say, with a modulo, Python will attempt to store that information as an integer until it is no longer able to.

4. Conditional Logic

Conditional logic is the idea that you can control the flow of a program by using special statements. You probably have heard of these special statements, even if it's not because of programming. For example, if you were giving someone directions to the grocery store, you might say that "if you see the donut shop, then you've gone too far."

Conditional logic when programming is very similar to real life, except that the syntax is much more locked down. Your conditions must adhere to the Python standard.

Whitespace is an important concept in Python, as it is how the language determines what statements are subordinate to others. In Python, the depth of a statement matters, but space between operators and comparators does not[4]. We encourage you to put space in between your operators and comparators to facilitate easier reading of code. Your style guide will dictate what kind of whitespace you should use, as there are several different methods. Some developers use two spaces, some use four spaces, some use tabs.

There are three big conditional statements in Python: `if`, `else`, and `elif`. Let's walk through them one-by-one.

If statements are the most basic of the conditional statements, but that doesn't mean that they're not as important. In fact, they are critical to controlling logic in the first place. For the sake of this section's simplicity, we will control most of our logic using integers, only dipping our toes into lists. Do note that it is possible to control logic using floats or even complex, but this is certainly more complicated.

First, let's declare two variables, `a` and `b`.

```
a = 0
b = 3
```

As we saw in section 1, Python will store both of these as integers. Now, we can ask Python if one of these integers is greater than the other.

```
if a < b:
  print('a less than b')
if b < a:
  print('b less than a')
```

[4] It is worth noting that whitespace does not matter nearly as much in some other languages, which instead use curly braces `{}` or a `begin` and `end` statement.

We would expect the output of this to only be "`a less than b`".

Let's look at each line of this script, line-by-line.

We begin our conditional statement with an `if` statement, and we provide a condition, whether `a < b`. The last thing in the line is a colon `:`. This is here to indicate to Python that this is an action that we want to carry out if the condition is met.

In the next line, we start off with whitespace. Like stated above, your whitespace is indicating to Python that this statement is subordinate to the conditional statement. This statement contains an executable action, a `print()` statement. This line will only be run if the condition that controls it is true. After the material inside of the `if` statement is run, the conditional section exits, and the program continues procedurally.

In line 3, we have another `if` statement. However, this time, we observe that the condition is not met. 0 is, in fact, not greater than 3. As a result, we skip anything inside of the conditional section.

We can also run conditional statements as boolean comparisons; Python can ask the question, "does the variable contain anything?" It returns `true` if something is contained, and `false` if the variable is empty, zero or `false`. For example, take the following lines.

```
if x:
  print('x has something')
if y:
  print('y has something')
```

In this case, the expected output is only "`y has something`". The variable `y` has a non-zero and non-false value, 3, so the `if` statement returns a `true`. Alternatively, the `x` variable has a zero-value, so the if statement returns a `false`, and the code inside of that statement is not run.

Python also has the ability to check equality, but it's important to not use the assignment operator. You must use a double equal sign `==` to compare the values. You can also run a non-equality using an exclamation mark and equal sign `!=`, a less than or equal to using a left chevron and equal `<=`, or a greater than or equal to using a right chevron and equal `>=`.

For this example, we will declare a third variable, `c`.

```
c = 3
```

Now, consider the following script.

```
if a == b:
  print('Condition 1 true')
if b == c:
  print('Condition 2 true')
if a != b:
  print('Condition 3 true')
if b <= c:
  print('Condition 4 true')
```

We would expect the following output.

```
Condition 2 true
Condition 3 true
Condition 4 true
```

Now that we've covered `if` statements, we can move on to different outcomes. Sometimes, we want to evaluate conditions and take one path if the condition is met, but also want to specify something to do if the conditions are not met. This is accomplished with an `else` clause. `else` statements must be prefaced with an `if` statement. Look at the following framework.

```
if condition == true:
  print('condition is true')
else:
  print('condition is false')
```

If the condition is true, the first suite is executed and the second is skipped. If the condition is false, then the first suite is skipped and the second is executed. Either way, execution resumes after the second suite. Like in an `if` statement, the code to be executed is defined by the indentation. This gives us a much tighter degree of control over the flow of our software. Let's look at our example from above.

```
if a == 0:
  print('a is zero')
else:
  print('a is not zero')
```

In this example, observe that our `else` statement has no switches on it. It's simply an action to be taken if the original condition is not met. In this way, writing `else a !=0:` would be syntactically incorrect. Let's look at a different way to implement a second degree of switching.

We can go one step deeper by looking at different steps of conditional logic that can offer different outcomes. Python implements this using `elif` statements. These statements must be prefaced with an `if` statement, and must contain some condition. Take a look at the following program.

```
a = 0
if a != 0:
  print('a is not zero')
elif a == 5:
  print('a is five')
elif a == 3:
  print('a is three')
elif a == 0:
  print('a is zero')
```

If you were to run this script, you would find that since a is zero, the first three conditions are not satisfied, but the fourth one is. If you want a high degree of granularity in your logic conditions, `elif` statements can offer this.

5. Loops

Loops are a critical part of any complex program. They allow you to specifically control the flow of your program without actually having to copy and paste the same code, multiple times. They rely on conditional logic to control them.

A word of warning when using loops: it is possible for loops to grow uncontrollably. If you do not implement your logic correctly, you could feasibly end up with an infinite loop. The only way to break this loop would be to terminate the program or wait for it to crash when it runs out of memory.

There are two fundamental loops in Python: `for` and `while`.

A for loop is an easy way to implement an iterative loop, and it works by examining some sort of numerical counting and using a conditional statement to determine whether a conditional state is true. In English, a for loop asks the condition, “for as long as the condition here is true, continue to do the loop”. Consider the following example.

```
for a in range(5):
    print(a)
```

This loop is declaring a variable `a`. By default, if a variable is not initialized, Python will auto-initialize the variable to be an integer 0. In the conditional statement, we’re asking Python whether the iterator, `a`, is within the range of numbers from [0,5]. If it is, we should print a. If it’s not, then we do nothing. Python’s default behavior is to auto-increment the iterating variable each time it runs the code in the loop. In this example, each time the condition in the loop is met and the loop runs, the value of `a` increases by one.

If we look at the output of this program, it shows up as the following.

```
0
1
2
3
4
```

You’ll notice that there are five numbers, but they start as 0. Why? Remember that our `for` loop looks at the iteration of the variable, starting at whatever value the iterating variable was initialized at. The variable `a` was initialized at 0, so the for loop starts at 0.

So why does the loop not include 5? Remember that a `for` loop only checks if the condition inside of the conditional statement is true. When we get to 5, the statement is no longer true; 5 is not between 0 and 5, since it the upper bound. Therefore, Python will break the loop and continue.

We can also define a more specific range of variables. Consider the following example.

```
d = 2
for d in range(2,9):
  print(d)
```

In this example, we're declaring an integer variable `d` and initializing it to be 2. Then, in our loop, we're iterating from 2 to 9.

Be careful to not confuse the condition with the loop, though. It is possible to put a boolean condition in that continues when the boolean is false. Remember that the loop looks at whether the condition itself is true, not the state of the condition. For example, look at the following code.

```
varsityAthlete = false
for varsityAthlete == false:
  print('not a varsity athlete')
  varsityAthlete = true
  print('condition changed')
```

In our example, `varsityAthlete` is a boolean variable, initialized to be false, but the condition of our conditional statement is true, therefore the code inside of the loop runs.

There is one final way to control the logic of your loops, `continue` and `break`. `break` will immediately break out of the loop and resume the program wherever the program continues after the loop, procedurally. It is an important tool in the developer's toolkit, but can be difficult to diagnose issues with. When debugging, be sure to put a `print()` statement just before you break so that you can determine whether your program is crashing or whether it's breaking deliberately.

The `continue` statement is of debatable importance, and some developers swear to never use it. A `continue` statement will not consider if something is a nested loop, instead allowing the program to recurse, if applicable. You should ask your instructor whether you should or should not use the `continue` statement.

For an example of using continue statements, consider the following code.

```
for testWord in words:
    if len(testWord) < len(longestAcceptableWord):
      continue
    if badLetters.search(testWord):
      continue
```

Our loop is using `testWord` in the list of `words` as an iterator. Inside of the loop, we have two conditional if statements. If the first condition is satisfied, then we continue out of the loop. If the first and second conditions are both not satisfied, then we can run some specified action.

Speaking of which, we can also use lists to iterate through a for loop, even if that list consists of strings. Instead of logically looking at the content of those strings, Python will just look at how many items there are in that string. Consider the puppies example from above.

```
puppyBreeds = ["doberman", "pug", "golden lab"]
puppyWeights = [5, 3, 3]
puppyTemperments = [3.22, 4, 5.5]

for x in puppyBreeds:
  print(x)
for y in puppyWeights:
  print(y)
for z in puppyTemperments:
  print(z)
```

We would expect this program to print the following.

```
doberman
pug
golden lab
5
3
3
3.22
4
5.5
```

Logically speaking, the `for` loop is asking the question, “is there another item in this list for me to go through?” In our puppies example, since there were three puppy breeds to go

through, we get to iterate through the loop three times. If we had five items in our list, we would iterate five times, as shown here.

```
jellicleCats = ["Mr. Mistoffelees", "Demeter", "Macavity",
"Skimbleshanks", "Rum Tum Tugger"]
for x in jellicleCats:
  print(x)
```

As you would expect, this prints out the following.

Mr. Mistoffelees
Demeter
Macavity
Skimbleshanks
Rum Tum Tugger

The `for` loop is only one kind of loop! There's also `while` and `do while` loops. The ideas behind these loops are similar, but their execution is a little bit different.

A `while` loop waits for a condition to be broken. For example, consider the following.

```
varsityAthlete = True
while varsityAthlete == True:
  status = raw_input('Are you a varsity athlete? Type yes or no: ')
  if status == yes:
    varsityAthlete = True
  if status == no:
    varsityAthlete = False
```

In this example, this (somewhat silly) program asks the user whether they are a varsity athlete. If they answer "yes," then the program will re-fill the variable `varsityAthlete` with `True`. If they answer "no," then `varsityAthlete` is `False`, the loop condition is broken, and the loop and script ends, since there's nothing after the loop.

`for` loops tend to be much more powerful and versatile compared to `while` loops, so you'll rarely see `while` loops.

6. Arrays

We covered the topic of lists in section 2, but in case you need a refresher, we can declare a list like shown in the following example.

```
residenceHalls = ["Lewis", "Gund", "Norton", "Mather", "McBride",
"Watson", "NCAs", "New Apts", "Leonard", "Hanna", "Old Kenyon",
"Tafts"]
```

In this declaration, we're filling the list `residenceHalls` with the names of the residence halls at Kenyon College.

Broadly speaking, a list is a type of array. You can think of arrays as tables in a spreadsheet. A list is a specific type of array, a one-dimensional array, or a table with only one row, and which stores only one type of data. What if we wanted to store more data, like another attribute? What if we wanted our single-row table to look more like a spreadsheet? Well, we can declare something called a two-dimensional array or a two-dimensional list. Take a look at the following code.

```
residenceHalls = [["Lewis", "Gund", "Norton", "Mather", "McBride",
"Watson", "NCAs", "New Apts", "Leonard", "Hanna", "Old Kenyon",
"Tafts"],["fresh", "fresh", "fresh", "fresh", "fresh", "upper",
"apartment", "apartment", "upper", "upper", "upper", "apartment"]]
```

In this example, we're still defining the names of residence halls, but we're assigning a second row of attributes, the type of residence hall. Python specifies that you enclose each dimension in square brackets `[]`.

Now we can store data in arrays, but how do we get that data out? Let's take a step back and look at one-dimensional arrays, or lists, first. Consider the puppy list from before.

```
puppyBreeds = ["doberman", "pug", "golden lab"]
```

We've filled an array of three things named puppyBreeds, but up to this point, we've only been able to put data in, not get data out.

Python allows us to extract data from our array according to the **index** of the entry. The index is an imaginary row in our imaginary table, and it specifies the column number in our spreadsheet analogy. Indices always start at 0, not 1. We can tell Python to print a certain object at a certain index. Consider the following example.

```
puppyBreeds = ["doberman", "pug", "golden lab"]
print(puppyBreeds[0])
```

In this example, our expected output is the following.

```
doberman
```

We can also extract the length of a list, using the `len()` function, which we can then use to iterate. The `len()` function always returns an integer. Consider the following code.

```
for i in range(len(puppyBreeds)):
  print(puppyBreeds[i])
```

Our expected output is the following.

```
doberman
pug
golden lab
```

Since there are three entries, our loop cycles through three times, and we get three outputs.

It is also possible to append an existing list. Take a look at the following code.

```
puppyBreeds = ["doberman", "pug", "golden lab"]
print(puppyBreeds[0])
puppyBreeds[0] = "pomeranian"
print(puppyBreeds[0])
```

Our expected output is the following.

```
doberman
pomeranian
```

We've appended our 0th entry in the list to be a pomeranian, instead of a doberman. Even though we're using the same print on the same index, we should get two different outputs.

Now, let's take it back to two-dimensional arrays. Consider the array `residenceHalls` that we declared above to be already declared in the following code.

```
print(residenceHalls[0])
```

You might think that this would print the first entry of both lists, `lewis` and `fresh`, but it doesn't. Rather, Python will treat the array with an index of 2 entries, and will print the entire first list, square brackets and all, and separated by the comma.

```
["Lewis", "Gund", "Norton", "Mather", "McBride", "Watson", "NCAs", "New
Apts", "Leonard", "Hanna", "Old Kenyon", "Tafts"]
```

The most logical way to separate this would be to put each array component into another list.

```
residenceHalls = [["Lewis", "Gund", "Norton", "Mather", "McBride",
"Watson", "NCAs", "New Apts", "Leonard", "Hanna", "Old Kenyon",
"Tafts"],["fresh", "fresh", "fresh", "fresh", "fresh", "upper",
"apartment", "apartment", "upper", "upper", "upper", "apartment"]]
hallNames = residenceHalls[0]
hallTypes = residenceHalls[1]
print(hallNames[0])
print(hallTypes[0])
```

The expected output of this should be the following.

```
Lewis
fresh
```

Like in a one dimensional list, we can iterate through this using loops.

```
residenceHalls = [["Lewis", "Gund", "Norton", "Mather", "McBride",
"Watson", "NCAs", "New Apts", "Leonard", "Hanna", "Old Kenyon",
"Tafts"],["fresh", "fresh", "fresh", "fresh", "fresh", "upper",
"apartment", "apartment", "upper", "upper", "upper", "apartment"]]
hallNames = residenceHalls[0]
hallTypes = residenceHalls[1]
for i in range(len(hallNames)):
  print(hallNames[i])
```

Python is also smart enough to know that if you change the value of something inside an array, that it should also change it in everything up the tree. For example, if I changed

`hallType[8]`, Leonard, to “fresh”, not only would it be represented as “fresh” in `hallTypes`, but also in `residenceHalls`.

7. Sets

Python sets are a data structure equivalent to sets in mathematics. It may consist of various elements, but the order of elements in a set is undefined. You can add or delete elements of a set. you can iterate the elements of the set, you can perform standard operations on sets (union, intersection, difference).

We define sets in Python similar to how we define single-dimension arrays. Consider the following example.

```
A = set('compsci')
print(A)
```

This should print the following.

```
{'c', 'o', 'm', 'p', 's', 'c', 'i'}
```

We can also use a for loop to iterate over the elements in a set. Consider the following example.

```
perfect = {6, 28, 496, 8128}
for j in perfect:
    print(j)
```

We might get the following output, though this might differ based on your Python interpreter. Again, sets aren't picky about the order that they're in.

```
496
28
6
8128
```

The `len()` function also works on sets. Consider the following code.

```
perfect = {6, 28, 496, 8128}
for j in perfect:
    print(j)
print(len(perfect))
```

This might output the following.

```
496
28
6
8128
4
```

For more details on the operations that can be carried out on sets, such as unions or intersections, I encourage you to view the Python documentation. This is beyond the scope of this text.

8. Functions and Classes

What if you had a set of ten lockers, and you wanted to assign some arbitrary value to each of those lockers, then print that value? Well, you could do this.

```
locker1-language = 'english'
locker1-owner = 'bob'
locker1-value = 3.37
locker2-language = 'german'
locker2-owner = 'sam'
locker2-value = 3.11
...
print(locker1-lanugage)
print(locker1-owner)
print(locker1-value)
...
```

However, you might notice that this seems really, really tedious. You're essentially typing the same frame of code a bunch of times. If only there were some easier way to code a repetitive action!

Functions

Well, as it turns out, Python does have such a way to do so: functions. A function in Python is just like a function in algebra. Let's take the example `y=mx+b`, a generic formula for a straight line.

Like above, if I wanted to calculate the Y value for 10 different slope/x/y-intercept combinations, I could do the following:

```
y1 = 3*6+8
y2 = 3*9+2
y3 = 6*2+2
...
print(y1)
print(y2)
print(y3)
...
```

Or, I could write a function that takes in three arguments, or variables that a function might use, and spit out some answers.

```
def linear(m, x, b):
    result = m*x+b
    return result
print(linear(3, 6, 8))
print(linear(3, 9, 2))
print(linear(6, 2, 2))
```

In the above code, I'm declaring a function called `linear()`. Linear does the repetitive math and spits out some response. As a result, I can write a lot less code. Let's break down each line of the function.

The first line of my function is called the function declaration. The declaration always begins with `def`, for define. This tells Python that you're about to create a function. Next, we have the name of the function. This is typically named according to what the function does, and, by convention, is typically written in `snake_case`. Finally, we have arguments that get passed into the function. These are the inputs that our function will manipulate. In this example, my function name is linear, and my arguments are m, x, and b.

The next line of the function is indented. This lets Python know that this material is inside of the function, and the enclosed code should be run when the function is called. Inside of the function, we can do anything we want. In this example, I'm doing some arithmetic: `m*x+b`. Then, I store the result of that in a new variable result.

Finally, I close my function with a return. The last line that is executed in your function should be a return. You can only return one thing, by default. You can return lots of different things: a `boolean`, a list, an array, a `string`, even a variable. In the example above, I am returning the variable result.

My function need not take in any arguments. Consider the following code.

```
def printaturtle():
    print('     _____     ____')
    print(' /      \  |  o | ')
    print('|        |/ ___\| ')
    print('|_________/     ')
    print('|_|_| |_|_|')
    return 0
```

When I call this function, I cannot pass in any other arguments. Consider the following.

```
printaturtle('turtle')
```

This is syntactically incorrect.

```
printaturtle()
```

This is the correct function call for creating an ASCII art turtle.

I know I said that you can only return one thing from a function above, but that technically isn't correct. Unlike algebra, you can actually return several things from a function or return nothing. For example, consider the following code.

```
def printaturtle():
    print('     _____     ____')
    print(' /      \\  |  o | ')
    print('|        |/ ___\\| ')
    print('|_________/     ')
    print('|_|_| |_|_|')
```

Even though I'm not returning anything, this is still syntactically correct for Python[5][6]. When I call this function, Python will just do the things inside of the function declaration, then return a 0 to indicate that the function ran successfully[7].

Now, consider the following.

```
def planets():
    return 'venus','jupiter'
```

If I were to call this function, I cannot simply put the return contents of this function into one variable. This is syntactically incorrect. That is, consider the following code.

```
def planets():
    return'venus','jupiter'
favoriteplanet = planets()
```

[5] Observe how all of the backslashes are doubled up. Recall how a newline uses the escape sequence \n. Likewise, \r does a carriage return and \t puts in a tab. Well, since \ is a special character, you need to put in an extra \ to ensure that the backslash itself is escaped.
[6] Python is somewhat special in this regard. Other languages, such as C++, require you to return something.
[7] The default behavior is to return a 0, but if your function is not syntactically correct or an exception is thrown, then your function will return 1 by default, or an exception, if specified. See Chapter 9 for exception handling.

This is not correct. The contents of the return of `planets()` has two values, but `favoriteplanet` is defined as a primitive data type.

We can get around this with a cheeky trick. Consider the following.

```
def planets():
    return 'venus','jupiter'
firstfav, secondfav = planets()
```

In this case, since venus is the first value that's being returned, it will be placed inside of firstfav. `jupiter` then gets placed in `secondfav`. This works because there are exactly as many elements in the contents of return as there are variables to put those contents into. Remember, Python does not throw away data. It would rather throw an exception and halt execution than throw away data.

Classes

We can also group functions together. Groups of functions are called classes. Right now, the difference between piling 50 functions at the beginning of your program and neatly breaking your functions into respective classes seems somewhat arbitrary, but I promise there's a reason.

Let's say I had a bunch of functions: `calculate_grade()`, `list_students()`, `delete_student()`, `breakfast_menu()`, `lunch_menu()`, and `account_money()`. I could break these into their own `classes`. Grades ought to go together, so we might put `calculate_grade()`, `list_students()`, and `delete_student()` in their own class called `grades`. Likewise, we could put `breakfast_menu()`, `lunch_menu()`, and `account_money()` into their own class called `food`. Consider the following code.

```
class grades:
    calculate_grade(name):
        grade = database_access(name)
        return grade
    list_students():
        students = []
        students = database_access_names()
        return students
    delete_student():
        remove_student(name)
        return 0
class food:
    breakfast_menu(date):
```

```
        return(breakmenu(date))
    lunch_menu(date):
        return(lunchmenu(date))
    account_money(name):
        return(acct_money_access(name))
```

While this program may seem trivial, you can see how grouping our functions could make things neater.

Now, let's imagine that you had a complex program that stored some sensitive information, like grades. You don't want your entire program to be able to access memory that contains the values of certain variables, like what grades belong to which students. So far, we've been able to place data into buckets, but we haven't learned how to put a lid or privacy screen on the bucket.

As it turns out, Python has a way of keeping data private. Just like how we have data types, such as ints, floats, and strings, we also have security types. Python has three distinct security types: public, private, and protected. A public variable can be accessed anywhere in the program. As long as you have the address of that variable, you can grab its contents. This is convenient, but not secure. You can protect an instance variable by adding a _ (single underscore) to the beginning of the variable. This effectively prevents the variable from being accessed, unless it is within a sub-class. Lastly, you can add a double underscore __ to make the variable private. This tells Python to keep this data private at all costs. Any attempt to touch the data by an unauthorized force results in an `AttributeError`. Consider the following code.

```
class student:
    def stugrades(database, name):
        db = create_database_conn(database)
        grade = get_grade(db, name)
        db_close(db)
        return(grade)
```

In this arrangement, anything can see the database, the name, and the grade. That's not very secure. Alternatively, we could try the following.

```
class student:
    def stugrades(database, name):
        _db = create_database_conn(database)
        _grade = get_grade(db, name)
        db_close(db)
        return(grade)
```

Now, if I tried to access the data for the database connection outside of this class, Python would not let me access that information. Similarly, consider the following code.

```
class student:
    def stugrades(database, name):
        __db = create_database_conn(database)
        __grade = get_grade(db, name)
        db_close(db)
        return(grade)
```

The double underscores mean that any attempt to read the contents of those variables, even by other functions inside of the class are effectively shut down by a program termination. The specifics of how classes manage public, protected, and private data are beyond the scope of this text.

Of note is, Python also supports self calls. Self calls are a little beyond the scope of this text.

Using External Modules

No programmer is going to write all of their code by themselves. A lot of solutions already exist for some pretty significant problems. So instead, programmers use libraries. A library is a piece of software that is written by a very experienced programmer (or development team), then is rigorously tested and released for other programmers to use. Python calls its code libraries "modules," and there are literally hundreds of thousands of modules available through two major package distributors: pip[8] and conda[9]. There are packages to do everything from display images to read and manipulate images, read CSVs and text files, perform computer vision tasks, parse webpages, and loads more!

Some common packages include `math`, `pandas`, `numpy`, `scipy`, `tensorflow`, `Pillow`, `os`, `sys`, and `platform`, and they expand the functionality of base Python.

It's not hard to use an external package. Most Python consoles have the ability to take a command line input to install a package. This command typically looks like the following.

```
pip install tensorflow
```

[8] You can search through the massive Python package library at pypi.org.
[9] Conda is the package manager for Anaconda, another integrated development environment.

You can then use that package to gain access to the modules included. For example, you might install the math package, then use some function inside of the math package. However, to use a package, you must first import it into your Python script.

```
import tensorflow
```

Python treats modules like classes. In reality, a module is just a wrapped class that's delivered to your computer ready to go. You can call a function inside of a module, just like you'd call any other function in a class.

```
tensorflow.Graph(argument1, argument2)
```

However, sometimes, we don't want to type out the entire name of the code module, especially if we use it a lot. So, there is a way to abbreviate your package name when you import it.

```
import tensorflow as tf
```

Now, my function call can be the following.

```
tf.Graph(argument1, argument2)
```

There are some established abbreviations for some Python packages. For example, `pandas` gets imported as `pd`, `numpy` gets imported as `np`, and `tensorflow` gets imported as `tf`. Most documentation gets written with these standard abbreviations, so you should use them whenever possible.

9. Exception Handling

While you're programming, things won't go according to plan; that's a given. However, a good programmer has a way of dealing with things when they go wrong. This concept is known as handling exceptions, and they're a great way to program.

Consider the following code.

```
def add(num1, num2):
    result = num1 + num2
    return result
add(1, 2, 3)
```

What if my user put in a third argument? Well, the Python syntax has been broken and I end up with an error message. We can deal with this, though. Consider the following code.

```
def add(num1, num2):
    result = num1 + num2
    return result
try:
    add(1, 2, 3)
except:
    warn("Something went wrong!")
```

Observe how we've added two new blocks: a try and an exception. Python will try to execute the code inside of the try block. If it cannot, then it will move onto the exception.

In this example, I've had Python output a pretty vague error message: "Something went wrong!" This is great if I want to test a user's input. For example, consider the following code.

```
def add(num1, num2):
    result = num1 + num2
    return result
try:
    num1 = int(input("Input a number: "))
    num2 = int(input("Input another number: "))
    print("Result: " + add(num1, num2))
except:
```

```
    warn("Did you put in integers?")
```

Now, if my user inputs in a float instead of an integer, when I try to run this program, instead of outputting a cryptic Python error message with a traceback, it asks the user if they put in integers.

Be aware that a try block must always be superseded by an except block. A try block by itself is syntactically incorrect.

What if we want to get the cryptic error message? Let's say we're running our try-except[10] block inside of a list loop, and some element in our list causes an error.

```
def add(num1, num2):
    result = num1 + num2
    return result
list = [1, 2, 3, 4, "Stinky Pete"]
for i in list:
    try:
        print(add(i+i))
    except:
        print("Something went wrong!")
```

We'd only get an error that something went wrong, not what element in our list caused an issue. While our list is small enough to notice, imagine that our list was thousands of elements longer.

Thankfully, Python has a way to deal with this.

```
def add(num1, num2):
    result = num1 + num2
    return result
list = [1, 2, 3, 4, "Stinky Pete"]
for i in list:
    try:
        print(add(i+i))
    except Exception as e:
        print("Something went wrong!")
        warn(e)
```

[10] Python refers to this as a try-except block. Other languages use terminology like try-catch, but they are the same thing.

If an exception is generated because something in the try block is syntactically incorrect, put that exception into the variable named e (a common standard for exceptions) and print a warning with that exception message.

Everything outside of the try-except block still gets run. Consider the following code.

```
def add(num1, num2):
    result = num1 + num2
    return result
list = [1, 2, 3, 4, "Stinky Pete"]
for i in list:
    try:
        print(add(i+i))
    except Exception as e:
        print("Something went wrong!")
        warn(e)
print("All done!")
```

As we can see, the final line falls outside of the try-except blocks, so even though the code inside of the try block will fail syntactically, the except block will handle the exception and allow the program to continue running.

10. Working with Files

As much as we'd like to pack everything into our Python scripts, sometimes, this just isn't reasonable. Let's say I wanted to analyze a dataset in Python about the responses that 16,182 people reported about how they remember the song "Jingle Bells, Batman Smells."[11] For one, that's a lot of data to put into Python by hand, and for two, you don't have to put it in all by hand. Instead, we can have Python read this file into memory so that it can manipulate it.

There are three main operations we can do to a text file: read, append, and overwrite. Reading is fairly straightforward: we read the file into memory and make some decision. However, the difference between writing and appending is a little bit more nebulous. Appending to a file opens that file as a *readable* file, but only *writes* to the end of that file. An appending operation cannot modify existing data. An overwriting or writing operation opens the file as a *writable* file, and it, by default, replaces and rewrites the new data over the old data. Writing operations are destructive!

For example, consider the demonstration data[12]. You can upload this to your repl.it or your working directory to use it. The demo data contains a header, plus the first hundred rows of a sample dataset. This would be a huge pain to input by hand into Python. Instead, we can use file reading to read the file into memory. Consider the following code.

```
file = open("soccer.txt", "r")
print(file.read)
```

Let's break down each line of this. The first thing I'm doing is declaring a variable that will hold the open file. The non-primitive variable type is a little bit too complicated to explain here. Next, I'm calling the function open(), which is a base function of Python. Next, I'm specifying the path of the file[13], along with the method: r for read, a for append, and w for overwrite. Finally, I'm printing the entire file.

The file variable contains the entire text file in memory, which I can then manipulate. For example, I can run the readline function. The readline function is a very handy tool for reading individual lines of a plaintext document. The syntax is provided below.

---

[11] This dataset actually exists. You can find it here: https://github.com/kim3-sudo/jinglebells/blob/master/jinglebells.csv

[12] You can find the downloadable demonstration data here: https://github.com/kim3-sudo/python-intro/blob/main/soccer.txt

[13] If you are running this on your local machine using software like Spyder, you probably need to specify a full, explicit file path: `C:\\Users\\<username>\\Downloads\\soccer.txt`. Depending on the Python interpreter, you may also need to specify that the file is in the current working directory: `./soccer.txt`.

```
file = open("soccer.txt", "r")
print(soccer.readline())
```

In fact, I can call readline multiple times to read multiple lines.

```
file = open("soccer.txt", "r")
print(soccer.readline())
print(soccer.readline())
print(soccer.readline())
```

The above example will read the first three lines. Likewise, I could loop through the entire file.

```
file = open("soccer.txt", "r")
for line in file:
    print(line)
```

When you finish with a file, you should *always* close it. This ensures that no other programs are trying to overwrite the same memory address at the same time.

```
file = open("soccer.txt", "r")
print(file.readline())
file.close()
```

Moreover, I can use another library to expand Python's functionality. One such library is the csv library. It allows us to manipulate a CSV natively. Consider the following code.

```
import csv
with open("soccer.txt", "r", newline = '') as file:
    reader = csv.reader(file, delimiter = ',', quotechar = '')
    for row in reader:
        print(', '.join(row))
file.close()
```

This example will print each line as a row in a plaintext output in your Python console. The csv module has some powerful parsing tools.

Likewise, you can write. Take the example above. We can adapt it to write another line to the file.

```
import csv
with open("soccer.txt", "a", newline = '') as file:
    writer = csv.writer(file, delimiter = ',', quotechar = '', quoting
= csv.QUOTE_MINIMAL)
    writer.writerow(['Sven Bender', 'Germany', 'NULL', 'NULL', 'Bor.
Dortmund', 'Res', '6', '07/01/2009', '2021', '81', '186 cm', '78 kg',
'Right', '04/27/1989', '27', 'CB/CDM'])
file.close()
```

Now, if you attempt to view the CSV in a program like Microsoft Excel, Libreoffice, or Google Sheets, you should see the new row has been appended.

Likewise, if you try to change the document mode from `"r"` to `"w"`, it would overwrite all of the existing data with just that one row and you'd have a pretty pitiful looking spreadsheet.