The Python Playbook

Introduction

This playbook serves as a starting point to jump into the Python programming language. This playbook is NOT a complete guide, and does not claim to be. Better people than me have written better comprehensive guides, but this is for those wanting to jump into the language and "lazy learn." Lazy learning is learning what you need when you need it, and not before. This makes solving problems more digestible early on and will give you a chance to shape your knowledge base to your specific needs. By the end of this, you should have enough of a basis of Python to search for problems, issues, or gaps in your knowledge and discover these things on your own. I may not introduce everything in a particular topic, but I may reference something I haven't taught later on. This is to encourage you to a) figure it out on your own or b) look it up for yourself. A big part of programming is looking through documentation, using Google to understand errors, and occasionally networking to solve problems. With that in mind, here's some places you can go to learn more about Python:

- https://docs.python.org/3/
- https://docs.python.org/3/tutorial/index.html

Data Types

Python uses what's called "duck typing." This means that you don't have to explicitly state the type of the variable you're using: Python will figure out what you meant. In Python, the basic data types you'll see are integers, floats, and strings. Integers and floats are numbers, except integers are whole numbers, and floats can have numbers after the decimal point. Because of this, integer math works differently. For example,

```
5 / 2 = 2.5
```

Is how we would see this happening, but in a computer between two integers, we get

```
5 / 2 = 2
```

This is weird, but makes sense: integers are whole numbers and can't have decimals. In Python, we realize most of the time we want floating point division, so there are two separate methods of division.

```
5 // 2 = 2
5 / 2 = 2.5
```

Variables are used to store values that we may want to use later. For example

```
value = 24
```

```
print(value)
--> 24
```

Now we can do a cool trick. The math operators of +, -, /, * can be used with the equals sign to increment, decrement, divide, or multiply a value.

```
value = 24
value -= 22
print(value)
--> 2
```

Conditionals

Conditional statements allow us to control the flow of our program. We can use values and variables to perform comparisons. The main statements we'll be working with are `if`, `else`, and `elif`. Example time

```
if 5 == 4:
    print("5 == 4!")
else:
    print("5 doesn't equal 4!")
```

If we were to run this program, we would expect for it to say "5 doesn't equal 4!" because the condition in the if statement isn't met. (Note: double equals == checks for equality while single equals = assigns a value to a variable. These CANNOT be mixed). What is executed is the `else` statement. This executes whenever the condition of the `if` statement isn't met. An `else` statement cannot exist without an `if` statement before it. Suppose we have this scenario

```
if 5 == 4:
    print("5 == 4!")
elif 5 == 5:
    print("5 == 5!")
else:
    print("5 doesn't equal 4!")
```

In this case, our output is "5 == 5!" and the `else` statement isn't executed. In an `if`, `elif`, and `else` block, only one of those statements can be executed. However, if we had something like this

```
if 5 == 4:
    print("5 == 4!")
else:
```

```
    print(“5 doesn’t equal 4!”)
if 5 == 5:
    print(“5 == 5!”)
```

You would get “5 doesn’t equal 4!” and “5 == 5!”. `If` statements operate independently of each other (on the same line). These `if` statements, however, do not operate independently of each other

```
if 5 == 5:
    if 4 == 4:
        print(“5 == 5 and 4 == 4”)
```

The outer **`if`** statement must be executed for the second **`if`** statement to be executed. We call the second **`if`** statement nested inside of the first one. Another way to handle nested statements is combining the two conditionals using an **`and`** or an **`or`** statement. In this case, we want an **`and`** statement

```
if 5 == 5 and 4 == 4:
    print(“5 == 5 and 4 == 4”)
```

An **`and`** statements needs both of the conditions to be true, while an **`or`** statement only needs one of the statements to be true.

Loops

Loops allow us to perform actions repeatedly. This is useful whenever we’re sorting or searching through items, reading a file, counting, etc. in Python, there are two main looping methods: **`while`** statements and **`for`** statements. These operate in a similar way, so let's show the differences

```
i = 0
while i < 4:
    i += 1
    print(“hello!”)

for i in range(4):
    print(“hello!”)
```

These statements perform the exact same task, however they’re formatted slightly differently. **`For`** statements generally have the value they’re using to iterate declared inside of them instead of **`while`** statements which need the value they’re using outside of the statement.

Data Structures

In Python, there are 3 main data structures: lists, tuples, and dictionaries. Lists and tuples both contain several values, however lists can be modified: tuples cannot.

A list: **`[1, 2, 3, 4]`**
A tuple: **`(1, 2, 3, 4)`**

You can iterate over a list (or tuple) using a for loop:

```
items = [1, 2, 3, 4]
for i in range(len(items)):
    print(items[i])
```

The **`len`** function (built into Python) can get the length of a list or a tuple. We can use brackets **`[]`** to get the item in the list at position **`i`**. In computer science, we start counting at 0, so **`i`** over the course of the loop will be **`0`**, **`1`**, **`2`**, and **`3`**.

Dictionaries are key-value stores (known as maps in other languages). These relate a particular value to another value (think... dictionary I guess). So for example, this is a Python dictionary:

```
data = {
    ‘banana’: ‘delicious’,
    ‘cake’: ‘so good’
    ‘dessert’: ‘best thing ever!’
}
print(data[‘banana’])
--> ‘delicious’
```

We’re creating a dictionary in Python, and assigning values to keys, and then looking up that key using square brackets to get the value of the key **banana**. Python dictionaries can store a variety of different types of data as values, such as strings, numbers, classes, and even functions.

Files

This is where your learning will get interesting. We’ll open up and read a file in Python, process it, and extract information. Here’s our sample code

```
with open(“filename.csv”, “r”) as f:
    for line in f:
        for value in line.split(“,”):
```

```
            print(value)
```

Okay, so a lot just happened. On the first line, we're opening up a file called "`filename.csv`" and we're operating in reading mode "`r`". Using a **`for`** statement, we can iterate line by line through the csv file. Then for each line, we split the line by commas, resulting in a list. Another way of writing this could be

```
f = open("filename.csv", "r")
for line in f:
    for value in line.split(","):
        print(value)
f.close()
```

So why the `f.close()`? The first method will automatically close the file at the end of the `with` statement. If we just assign it to a variable, then we need to close it at the end. So if we had a file that looked like

```
Cake,24,47,33
Potato,22,21,20
```

Our program would put out

```
Cake
24
47
33

Potato
22
21
20
```

So you may be wondering why there is a gap between **`33`** and **`potato`**, and that's because of what's called the newline character **`\n`**. The newline character is what the computer uses to represent someone typing the enter key. So we just need to rip out the newline character

```
with open("filename.csv", "r") as f:
    for line in f:
        for value in line.strip("\n").split(","):
            print(value)
```

There we go! Now we've stripped out the newline character.

Functions

So you may have noticed we've been using these words with parentheses after them. These are called functions. Functions are reusable pieces of code that performs a task and can return something from them. We can write our own functions

```
def my_function():
    return 24

value = my_function()
print(value)
> 24
```

We use the **`def`** keyword to define a function, and then call it by using its name and curly brackets. The **`return`** statement in a function can return a value, but it doesn't always have to. Whenever the **`return`** statement is used, however, it will immediately end the function that it's in. This is generally why it's located at the end of a function. Now that I've said that, time to break that rule

```
def greater_than_zero(value):
    if value > 0:
        return true
    return false

if greater_than_zero(4):
    print("4 is greater than zero!")
if not greater_than_zero(-4):
    print("-4 is not greater than zero!")
```

By returning **`true`** and **`false`** (otherwise known as boolean values), the if statements can be evaluated as true when the value **`4`** is passed to the **`greater_than_zero`** function. With functions, we can reuse code over and over again without having to repeatedly write it.

Example Programs

```
import random
import sys

my_cards = 0

def did_win(value):
      rand_val = random.randint(1, 21)
      print(f"your opponent scored {rand_val}")
```

```
        if value - rand_val >= 0:
            return True
        return False

for i in range(20):
        new_card = random.randint(1,14)
        print(f"new card is: {new_card}")
        response = input("will you take it? answer 'yes' or 'no': ")
        if response == 'yes':
            my_cards += new_card
        if my_cards > 21:
            print("You got higher than 21! You failed!")
            break

        print(f"your total score now is {my_cards}")
        response = input("are you done?: ")
        if response == 'yes':
            print(f"you scored {my_cards}")
        if did_win(my_cards):
              print("congratulations! You won!")
        else:
              print("Oh no! You lost!")
        sys.exit()


print("game over!")
```

So we've got some new goodies in this blackjack program. We have the **`input`** function, which will print something to the screen but will wait for the user to type something and then hit enter. We also have some string interpolation, which allows us to embed numbers into a string. Python has multiple ways of embedding variable data inside of a string, but this is the most common way.

We're also importing **`sys`** and **`random`**. These are packages standard to Python, and they perform specific tasks for us so we don't have to implement them ourselves. The **`sys`** package gives us tools related to the system (getting environment variables, starting/stopping the current program, etc) and the **`random`** package gives us tools to use to randomly generate numbers and choices.

List Comprehension

This is typically one of the more confusing parts of Python for people. List comprehensions are a way to create lists of items in one line. This is extremely useful if you need to package and format data in a different manner and send it out in some sort of payload (example:

converting Python objects into JSON). So lets look at some examples of a list comprehension:

```
data = [x for x in range(10)]
print(data)
--> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

So, what exactly happened here? We used a for loop, iterated through numbers 0-9 (for a total of 10 numbers) inside of some brackets. But why exactly did we get these numbers? We're actually pulling out each individual number from range, assigning it to variable x, and outputting it into the list. Here's a more interesting example:

```
data = [x + 1 for x in range(10)]
print(data)
--> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Hopefully this makes things a little more apparent. Whatever is directly on the left side of the for loop is what is getting put into the array, and the variable **x** is what we're pulling out of **range**.

Classes

Classes are essentially just containers that hold variables and functions within themselves. When creating an instance of a class, this is called an object. You can have multiple objects of one class. This is a topic that is extremely complicated and there are entire books based on this topic, so we're going to keep this simple.

```
class Book:
    def __init__(self, author=None, content=None):
        self.author = author
        self.content = content

    def print_book(self):
        print(f"{self.author} => {self.content}")

book = Book(author='john', content='amazing exceptional content')
print(book.author)
-->  john
book.print_book()
--> john => amazing exceptional content
```

The **__init__** function is what's used to initialize an object. An important thing to note is that if you want to use a function in a Python class that accesses data inside of a class, you

need to pass the **`self`** argument to the function. You'll also see that in the argument list for the initialization function that we have **`author=None`** which is a way for us to set a default value in case one is not passed in.

One thing you'll notice if you're coming from other languages is not mention of public or private keywords: this doesn't exist in Python. You can always access the individual variables inside of an object, so there's really no way to "protect" a variable or function.

Advanced Topics

You'll find very limited discussion from me in these topics as if you're looking for these particular resources/aspects of Python, then you're probably good enough to solve these problems on your own. I'll point you directly to the documentation for this, and give you a brief overview of what this looks like in Python.

Multiprocessing

One of the main complaints of Python is having to deal with the global interpreter lock (GIL). I'll let you look up exactly what this is, but it essentially means since Python is a scripting language that's interpreted, you can't really get true parallelism in the same process. So we can sidestep this by creating multiple processes, or instances of our program. This allows us to saturate the number of cores on our machine to enable greater utilization of system resources.

- https://docs.python.org/3/library/multiprocessing.html

Asynchronous Programming

Python has a very weird relationship with asynchronous programming, but has become a bit more stable and usable in Python3.6+ using the **`async`** and **`await`** syntax from the **`asyncio`** package. This is particularly useful in IO bound tasks (reaching out to a network service, reading a file from disk, waiting for something to respond, etc.).

- https://docs.python.org/3/library/asyncio.html

Dealing with the "Slowness" of Python

People will complain and bash Python for being a "slow" language and that you can't accomplish heavy computational tasks with it. Python is by nature going to be slower than C/C++ in terms of execution since it is an interpreted language, but most people understand this and have developed work arounds. Python has a LOT of bindings for C/C++ libraries, allowing you to execute code from those languages inside of your Python programming. What this means is that programming using Python code can be slow, but programming in Python isn't necessarily slow. If you're needing to do some serious number crunching, the first place you should reach out to is NumPy.

- https://numpy.org/

Machine Learning

I think Python probably has one of the most pleasant ways to perform machine learning tasks if you just need to get something done. There are a lot of packages you can use with a variety of different models/methods, so it's really up to your needs and how deep you want to go.

- https://scikit-learn.org/stable/
- https://keras.io/
- https://www.tensorflow.org/

Duck Typing is Stupid! Where is My Static Typing?

Some people claim that dynamically typed languages will introduce more bugs. Well, recent versions of Python allow optional type declarations for functions and return values. There are also packages (such as Pydantic) that will give you even more typing support and data validations.

- https://docs.python.org/3/library/typing.html
- https://pydantic-docs.helpmanual.io/

How Should I Do Testing?

Everyone has their opinions on testing and how/when you should do it. A lot of people don't like the interpreted nature of Python because you won't know how a particular function will work until it's used. I would argue if this is your complaint, you're probably not doing any test driven development/unit tests and just relying on your compiler to tell you what you did was wrong. I honestly think this is more dangerous than using an interpreted language (if you're using Rust, yes the compiler is very helpful, but it won't catch logical issues so you still need tests). I'd recommend using the pytest package.

- https://docs.pytest.org/en/stable/

Web Development

This is a gigantic can of worms and I'm gonna try to leave this as open ended as possible. There are LOTS of Python web frameworks prioritizing various different aspects. I cannot list all of them and tell you which ones are the best. If you're just starting out, something like Flask or Django should be sufficient. If you're feeling more adventurous, you can jump into asynchronous frameworks such as FastAPI or Quart (basically Flask, just async). There are also various tools that are used alongside these frameworks, so it's up to you to decide what you want/need. There's plenty of support for websockets and 'realtime' stuff if that's your thing.

- https://www.djangoproject.com/
- https://flask.palletsprojects.com/en/1.1.x/
- https://fastapi.tiangolo.com/
- https://gitlab.com/pgjones/quart
- https://gunicorn.org/

Automation

I think one of the bigger appeals of Python is automating various tasks, from devops to just sending emails. There are a number of tools you can use to accomplish these various tasks, so I'm going to link a commonly recommended automation course/website. This will get you up to speed pretty quickly with Python and using it to accomplish some common tasks you may face in your day-to-day life.

- https://automatetheboringstuff.com/