Chapter 1

Python

Python Comments are #great

Kliff

The Python programming language was created by Guido Von Rossum many moons ago around the 1980s with its first release in 1991. Since then, there have been various changes to the language, some of which are interesting and others which are not as interesting. It was named after the British comedy group Monty Python. For some buzzwords, we can say Python is a high-level, dynamically typed, garbage-collected, primarily imperative programming language. You may also hear the terms "duck typing", "scripting", and "interpreted". What do all of these weird terms mean? I am sure someone knows.

If you want to follow along through this chapter, you can run the files with **`python file.py`**. Alternatively, you can also use the repl (Read-Eval-Print-Loop). Just type `python` in your terminal. (At the time of writing, I am using python 3.8)

1.1 Introduction

Unlike the previous languages you have seen in 131/132 and 216, Python does not use a compiler, so no machine code is generated. This means that compile-time checks do not exist. This basically means that every check or error is done during run-time. I'll expand more on this in a bit, but for now, let's write our first Python program.

```
1 # hello_world.py
2 print("Hello World")
```

Despite this being very simple, we've already learned several things.

- Single-line comments are started with the pound or hashtag symbol
- no semicolons to denote the end of the statement (what does this imply?)
- `print` is used to print things out to stdout
- functions use parenthesis (weird we need to say this)
- python file entension is `.py`
- Strings exist in the language (most languages have them, but some do not)

Can you think of more?

Now to run our program we can just use

```
1 python hello_world.py
```

Congrats, you have just made your first program in Python!

1.2 Typing

Now we just said that Python does not use compile-time checks, and all checks and errors are done at run-time. Let's see this in action.

```
1  # program1.py
2  variable1 = 5
3  variable1 = "hello"
4  variable2 = 4
5  variable3 = variable1 + variable2
6  print(varable3)
```

Here is a simple program that sets some variables, adds two things together, and then prints the results. This looks weird, but first, let's run and see what happens, and then we can look at the syntax.

```
python program1.py
TypeError: can only concatenate str (not "int") to str
```

Looks like we have an error! Seems like `variable1` and `variable2` have different types, so we can't use `'+'` on them. This is interesting for 2 main reasons:

- We did not get an error on line 3 when we change `variable1` from an `Integer` value to a `String` value
- There is no automatic conversion of integers to strings as we saw in Java

The first point is really important, but before we talk about it, let's just modify our code so that it runs without any errors by deleting line 3.

```
7  # program1-1.py
8  variable1 = 5
9  variable2 = 4
10 variable3 = variable1 + variable2
11 print(varable3)
```

Now if we run our code, we get a different error. We get **`'NameError: name 'varable3' is not defined'`**. Notice that because we check everything during run-time, this error was not picked up until Python was about to execute it. **Many bugs from beginner (and experienced) Python programmers are due to misspelled variable names**. Here is a more visual demonstration of run-time erring.

```
12 # program1-2.py
13 variable1 = 5
14 variable2 = 4
15 variable3 = variable1 + variable2
16 print(variable1,end="") # no new line when printing. What can you infer about this?
17 print(" + ",end="")
18 print(variable2,end="")
19 print(" = ",end="") # gross. We will learn to convert later
20 print(varable3)   # still misspelled
```

If we run the above code, we get **`'5 + 4 = '`** printed out, and then we get the same **`'NameError'`** as before. An error-free program would look like

```
21 # program1-3.py
22 variable1 = 5
23 variable2 = 4
24 variable3 = variable1 + variable2
25 print(variable3)
```

Now that we fixed this issue, we can talk about why we didn't get an error on line 3 in `program1.rb`.

Notice that in the above code, we set values to variables, but we didn't define the type like we did in Java and C. This is because of Python's **type system**. A language's type system determines how data and variables can be used. Some key

words that may describe a type system could be dynamic vs static or manifest vs latent. You may also hear of type safety and strong vs weak. To determine how data and variables must be used, rules are made and enforced by what is typically called a type checker. Ultimately: How does a language know if a variable or piece of data is an `int` or a `pointer`? It does so by type-checking.

Python uses both dynamic and latent typing in its language. Dynamic type checking is a form of type checking which is typically contrasted to **Static type checking**. For the most part, you probably only used static type checking since both C and Java are statically typed. **Static typing** means that the type of a variable, construct, function, etc is known before the program runs. It can also be said that type checking is run at compile time. Should we then use a type in an incorrect manner (as we did in `program1.py`), then the compiler will raise an error and compilation will be aborted. Contrasted to **Dynamic typing** which means that the type is only calculated at run-time. (Since Python has no compiler, it cannot be statically typed). Consider the following:

```
1 # err.c
2 void main(){
3     int x;
4     x = "hello"
5 }
```

Should we try to compile this code with the `-Werror`[1] compile flag, we will get the following: **`'error: assignment to 'int' from 'char *' makes integer from pointer without a cast'`**. This is because during compilation, the compiler marks the x variable as having a type of int, yet it is being assigned a pointer.

Now, how did gcc know that there was a type issue here? The first conclusion would be that we declared `x` as an `int` explicitly on line 3. This is called **manifest or explicit typing**, where we explicitly declare the type of any variable we create. This is in contrast to **latent or implicit typing** where we don't have to do this as we saw in python. It is important to note: **manifest typing is not the same as static typing**. We will see this in OCaml as the language is statically typed, but uses latent typing for its variables.

Back to Dynamic type checking, let's look at the following:

```
1 # checking.py
2 def add(a,b):
3     print(a + b)
4
5 add(1,2)
6 add("hello"," world")
```

To begin, this is how you create a function in Python. Functions begin with the `def` keyword, and the body will always be indented. We will go into Python code examples later, so for now, just know this is a function that takes two arguments and then prints the result of `a + b`. Since Python is dynamically typed, types aren't assigned to the parameters `a` and `b` until we run our code, and not until the values are actually used. This allows us to call `add` with both `Integers` and `Strings`. Much like before, it is important to note that **latent typing is not the same as dynamic typing**.

In any case, you may be wondering how Python knows the types of variables if we don't explicitly declare their types. The process of deciding a type for an expression is called **Type inference** and we will go more in-depth with this in the OCaml section, but there are many ways that type inference can be done. In fact, you already saw one way with our `err.c` program. We did not say that `"hello"` was a `char *`, yet gcc knew because of the syntax of the datatype. The same holds true for Python. Integers are numbers without decimal points. Floats are numbers with decimal points. Strings are anything put in quotes. You can check this with the `type` function. For instance, try `type("hello")`.

1.3 Python Scoping

When we think of a scope, we think about where in the code can we use something, whether it be a variable or a piece of data. Consider the following:

[1] Canonically it will just raise an error and still compile

```
1  # scoping-1.py
2  a = 5
3  b = 20
4  def h(c):
5    d = True
6    e = 0
7    while d:
8      if c < b - c:
9        c = c + 1
10       e = e + 1
11     else:
12       d = False
13   return e
14
15 f = input("Enter a number: ")
16 print(h(int(f)) + a)
```

While this program has a ton of new syntax that we have not seen in python before, I believe we can take a look at this program, take a few notes, and make some hypotheses about some things we can and cannot do in python. You can check if your hypotheses are correct in the syntax part of this chapter [2].

In any case, we can ask ourselves: where can each variable be used? Do you believe you could use a from lines 2 - 16? Can c be used outside of lines 4 - 13? While these questions may have somewhat intuitive answers based on your past programming knowledge, you then have to consider: why does asking this question matter?

Consider the following:

```
1  # scoping-2.py
2  a = 5
3  b = 20
4  def h(a):
5    d = True
6    e = 0
7    while c:
8      if a < b - a:
9        a = a + 1
10       e = e + 1
11     else:
12       c = False
13   return e
14
15 f = input("Enter a number: ")
16 print(h(int(f)) + a)
```

Here, I changed a variable name, and we now need to consider the scope of the a variable and if this impacts the outcome of the program. In this case, no change to the program occurs, but what about something simple like the following?

```
1  # scoping-3.py
2  a = 5
3  def h():
4      print(a)
5  h()
```

[2] Whenever you learn a new language, a great way to pick it up quickly is to analyze a snippet of code and mark down what you think everything is doing. Then, make a hypothesis about what you think the code will do, and then run it. You will either affirm your assumptions and enforce your belief about how the language works, or you get something you did not expect. When this happens, this means there is a gap in your knowledge. You can now make assumptions about why the outcome occurred and what you didn't consider. The best thing to do here is to modify the code, make a new hypothesis, and continue this process to learn more. For example: I see that `True` and `False` exist and are capitalized here. I have two hypotheses: 1: booleans exist in the language. 2: You must capitalize true and false in the language. Now to test this: a simple `type(True)` tells you this is of type bool, which means python calls them bool and not boolean. Additionally, `type(true)` gives an error, which affirms the hypothesis that `bool`s must be capitalized.

Where can we use the a variable? If I modify this just a tad bit more:

```
# scoping-4.py
a = 5
def h():
    a = a + 1
    print(a)
h()
```

What does your intuition say? What does `python scoping-4.py` say? In this case, we get an error! Wild. Why is this?

Typically, global variables are used throughout the entire program, and if one part of the program can modify that variable, it would impact other parts of the program that use that variable. Thus, making global variables read-only is good practice. In this case, they are typically called global constants, or just constants. Since Python doesn't need a main function (but it can have one), any variable defined outside a function is global by default.

If we wish to modify a global variable, we can do so using the `global` keyword.

```
# scoping-5.py
a = 5
def h():
    global a
    a = a + 1
    print(a)
h()
```

So what happens if we have a local and global variable of the same name, like we did in `scoping-2.py`? Can we edit the global variable and not the local one?

```
# scoping-6.py
a = 5
def h():
    a = 5
    global a
    a = a + 1
    print(a)
h()
```

Here we get an error: a python function seems to only be able to deal with either a local variable or a global variable, not both. But notice the wording of the error message: `"name 'a' is used prior to global declaration"`. This seems to imply that you can declare a global variable inside a function?

```
# scoping-7.py
def h():
    global a
    a = 6
h()
print(a)
```

This also seems a bit counterintuitive from what we know of our past experience with programming languages. Nonetheless, this is how Python operates, and we must keep this in mind if we continue to use this language. These scoping rules also seem to impact what functions refer to.

```
# scoping-6.py
a = 5
def h():
    print(a)
h()
a = 6
h()
```

Notice here that this prints 5 and then 6. Not all languages have this behavior (like OCaml), and this becomes important when we talk about `closures`.

Consider the following and think about what this means when designing python programs:

```
# scoping-7.py
if True:
    b = 5
else:
    b = 6
print(b)
```

1.3.1 Nested Functions

In the same vein as scoping, we can define functions within other functions.

```
def outer_func(a):
    def inner_func(b):
        return a + b # can access 'a' in the inner scope
    # could not access 'b' here in outer scope
    return inner_func(4)

print(outer_func(5)) # prints 9
```

The inner function here can access all variables in the `outer_func` scope, but the reverse is not true. A common use case of this is when using higher order programming (a future chapter).

1.4 Object Oriented Programming

Python supports the Object Oriented Programming paradigm, which means we can use some of our java knowledge when working with classes. Let's first look at our syntax though:

```
# poop-1.py python object oriented programming
class Square:
    def __init__(self, size):
        self.size = size

    def area(self):
        s = self.size
        return s * s

s = Square(5)
print(s.area())
```

What looks new to you here? Perhaps some things to note:

- `class` keyword looks similar to java
- `__init__` must be a constructor of some sort
- `self` is a parameter and seems to be similar to java's `this`
- no new keyword when making a new object
- methods are defined inside the body of the class
- methods are called using the dot (.) syntax

- Much like functions, it seems like indentation matters here. We said earlier that the body of a function will always be indented. Additionally, we said there were no semicolons. Notice there are also no brackets either. This means Python must need some other way to figure out which lines of code are associated with each other. In this manner, new lines and indentation matter quite a lot in Python.

Noticing these things may raise some more questions. What happens if we want to inherit from a parent class? How exactly does `self` work? Is there a default `toString()` method?

To inherit from another class, we need to add a parameter to the class declaration:

```
1  # poop-2.py python object oriented programming
2  class Rectangle:
3      def __init__(self,width,height):
4          self.width = width
5          self.height = height
6
7      def area(self):
8          return self.width * self.height
9
10 class Square(Rectangle):
11     def __init__(self,size):
12         super().__init__(size,size)
13
14     def __str__(self):
15         return "Width: " + str(self.width) + "\tHeight: " + str(self.height)
16 r = Rectangle(4,5)
17 s = Square(5)
18 print(r.area())
19 print(s.area())
```

Notice there is a built-in `toString` method called `__str__`, and it can be overwritten. Notice that `super` is still a thing, and we can use it to call the parent's constructor. Lastly, notice that `self` is not explicitly passed in, but you can use it to refer to the current object's attributes.

1.5 Syntax

Here are just basic syntax things for the python language. At the end of the day, a for loop is a for loop, an array is an array. These are just simple syntax notes about these things. I will say that most languages will always have something of the following:

- Built-in data types/structures
- Control flow structures (for, if, while, jmp)
- I/O (print/read)
- variable assignment and manipulation
- functions
- comments

Knowing the syntax is important to learning a language, but it is also important to know the lingo a language uses. For example, Python does not have `booleans`, but rather they have `bools`. This may be pedantic and may sound pretentious to some (which is not entirely false), but consider:

```
1  # syntax.py
2  def f(x):
3      if type(x) == bool:
```

```
4         return 3
5     else:
6         return 4
```

Having line 3 be `if type(x) == boolean` would fail due to `boolean` not being something in the python language. Just keep this in mind as you learn more languages.

1.5.1 Data Types and Structures

Python has a few common data types built into the language. The typical suspects consist of `ints`, `floats`, `bools`, and `strs`. What may be shocking is that there is no `char` type in Python. Some examples:

```
1  1                    # int
2  1.2                  # float
3  1.5 * 2              # float
4  7//2                 # int
5  7/3                  # int
6  int(1.5)             # int
7  "hello"              # str
8  len("hi")            # int
9  "a" + "b"            # str
10 "na"*15 + "batman"   # str
11 True or False        # bool
12 True and False       # bool
13 not False            # bool
```

From previous languages, we are (hopefully) familiar with arrays and hashmaps. In python, we have lists and dictionaries. They also support tuples and sets(!).

```
1  a = [1,2,3]          # list
2  [1,"hi",True]        # list
3  sorted([3,2,1])      # list
4  a[0]                 # returns 1
5  a[-1]                # returns 3
6  a.append(4)          # list becomes [1,2,3,4]
7  a[1:]                # returns [2,3,4]
8  a[:2]                # returns [1,2]
9  a[1:3]               # returns [2,3]
10 a[1:3:-1]            # returns [3,2]
11 a[::-1]              # returns [4,3,2,1]
12 [1,2,3] + [4,5,6]    # returns [1,2,3,4,5,6]
13 a = {"key":"value",1:True} # dict
14 a["key"]             # returns "value"
15 a[1]                 # returns True
16 (1,2)                # tuple
17 (True,"Hello")       # tuple
18 (True,2.4,"hi")      # tuple
19 {1,2,3}              # set
20 {1,2,3,3}            # set that is just {1,2,3}
21 {"hi",1,False}       # set
```

1.5.2 Control Flow

It is always helpful to be able to control the order in which commands are executed, conditionally or unconditionally. In fact, it is one of the requirements for Turing completeness (something covered in a later chapter). The classic types of control statements are looping constructs and conditional execution (if, else).

Some looping constructs:

```
# conditional
if 3 > 4:
    print("False")
elif 4 > 5:
    print("Also False")
else:
    print("True")

# while loop
a = 0
while True:
    if a % 3 == 0:
        print("three")
    elif a % 2 == 1:
        continue
    elif a % 4 == 1:
        break

# for loop
for x in ["a","ab","abc"]:
    print(len(x))

for x in range(3):
    print(x)
```