# 2. Python Basics

In this chapter, we get started with very useful Python basics.

## 2.1. Comments

A comment is a piece of text that is not executed within a program. It can be used to provide additional information to help with code comprehension.

A comment is started with the # character and continues until the end of the line.

```
# This is a comment line.
```

## 2.2. Whitespace Formatting

Curly brackets are used to separate code blocks in many languages. Indentation is used in Python: This makes Python code very readable, but it also implies that formatting must be done carefully.

```
for i in [1,2]:
    for j in [1,2]:
        print(i+j)
```

```
2
3
3
4
```

Inside parentheses and brackets, whitespace is ignored, which is useful for long-winded calculations:

```
my_list = [1, 2, 3]
my_list_of_lists = [[1,2],
                    [3,4]]
```

## 2.3. Python Data Types

In Python, every value is referred to as a "object." And each object has its own data type. The following are the three most common data types:

### 2.3.1. Integers

Integers are integer numbers that can be used to represent objects such as "number 8."

```
# Example integer numbers

a = 2
b = 1
c = -1
```

### 2.3.2. Floating-point numbers

Floating-point numbers are a type of number that can be used to represent real numbers or floating-point values.

Contents

```
# import the math module
import math

# Code to compute solution to the quadratic equation of the form a x^2 + b x + c = 0
sol1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a)
sol2 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a)
```

```
print([sol1,sol2])
```

```
[0.5, -1.0]
```

The build-in Python `type()` function returns the type of these objects.

```
type(sol1)
```

```
float
```

## 2.3.3. Strings

A string is used to define a sequence of characters. Consider the word “hello.” Strings are **immutable** in Python 3. You can’t modify it afterwards if you’ve previously defined one.

```
my_string = "Hello world"
my_string.replace("world", " Mahidol")
```

```
'Hello  Mahidol'
```

```
print(my_string)
```

```
Hello world
```

While commands like `replace()` and `join()` can modify a string, they generate a copy of it and apply the changes to it rather than rewriting the original.

### 2.3.3.1. How to Create a String in Python

Using single, double, or triple quotations, you can make a string in three different ways. Here’s an example of each possibility:

```
second_string = 'Mahidol University'
```

The `print()` function can then be used to print your string in the console window. This allows you to go through your code and make sure everything works properly. Here’s a sample for you:

```
print(second_string)
```

```
Mahidol University
```

```
third_string = '''Department of Mathematics,
Faculty of Science,
Mahidol University'''
```

```
print(third_string)
```

```
Department of Mathematics,
Faculty of Science,
Mahidol University
```

### 2.3.3.2. Concatenating strings

The next skill you can learn is concatenation, which is a method of joining two strings using the "+" operator. Here's how you do it:

```
my_string + " from " +  second_string
```

```
'Hello world from Mahidol University'
```

Note that the + operator cannot be used on two different data types, such as string and integer. You'll get the following Python error if you try it:

### 2.3.3.3. Escaping Characters

In a Python string, backslashes (\) are used to escape characters.

For example, the given code can be used to print a string containing quote marks.

```
# Quote from Albert Einstein

"\"Imagination is the highest form of research\". - Albert Einstein"
```

```
'"Imagination is the highest form of research". - Albert Einstein'
```

### 2.3.3.4. String Indexing and Slicing

Because strings are lists of characters, Python strings can be indexed using the same notation as lists. Bracket notation (`[index]`) can be used to access a single character, or slicing can be used to access a substring (`[start:end]`).

Indexing with negative numbers counts from the end of the string.

```
print(my_string[0])
print(my_string[0:5])
print(my_string[-5:])
```

```
H
Hello
world
```

### 2.3.3.5. Iterate String

To iterate through a string in Python, "`for…in`" notation is used.

```
for c in second_string[:8]:
    print(c)
```

```
M
a
h
i
d
o
l
```

# 2.4. Python Variables

Variables are containers for storing data values. Variable names are case-sensitive.

Variables in Python 3 are special symbols that assign a specific storage location to a value that's tied to it. In essence, variables are like special labels that you place on some value to know where it's stored.

The code below demonstrates how to store a string in a variable.

```
my_string = "Hello world"
```

Let's break it down a bit further:

- my_string is the variable name.
- = is the assignment operator.
- "Hello world" is a value you tie to the variable name.

Variables do not need to be declared with any particular type, and can even change type after they have been set.

```
x = 10       # x is of type int
x = "SCMA"   # x is now of type str
print(x)
```

```
SCMA
```

## 2.4.1. Variable Names

A variable can have a short name (such as x and y) or a longer name (such as age, carname, or total volume). Variables in Python have the following rules:

- The name of a variable must begin with a letter or the underscore character.
- A number cannot be the first character in a variable name.
- Only alpha-numeric characters and underscores (A-z, 0-9, and _) are allowed in variable names.
- Case matters when it comes to variable names (age, Age and AGE are three different variables)

## 2.4.2. Casting

Casting can be used to specify the data type of a variable.

- `int()` function trims the values after the decimal point and returns only the integer/whole number part.
- `float()` function is used to convert any data type to a floating-point number.
- `str()` function is used to convert integer into a string.

```
x = str(10)
type(x)
```

```
str
```

```
y = float(10)
type(y)
```

```
float
```

```
z = int(10.1)
z
```

```
10
```

# 2.5. Python Lists

In Python, lists are another important data type for specifying an ordered series of elements. In particular, they allow you to group related data and perform the same operations on multiple variables at once. Lists, unlike strings, are mutable (that is, they may be changed).

Each value in a list is referred to as an item, and it is enclosed in square brackets `[ ]`, separated by commas. It is good practice to put a space between the comma and the next value. The values in a list do not need to be unique (the same value can be repeated).

Empty lists do not contain any values within the square brackets.

```
my_list = [1, 2, 3]
```

Alternatively, you can perform the same thing with the `list()` function:

```
third_list = list((1 , 2, 3))
print(third_list)
```

```
[1, 2, 3]
```

```
my_list == third_list
```

```
True
```

In Python, lists are a versatile data type that can contain multiple different data types within the same square brackets. The possible data types within a list include numbers, strings, other objects, and even other lists.

```
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]
```

## 2.5.1. How to add items to a List

You can add new items to existing lists in two methods. The first involves the use of the `append()` method:

The `insert()` method can be used to add an item to the specified index:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

```
[1, 2, 3, 4]
```

```
my_list.insert(2, 2.5)
print(my_list)
```

```
[1, 2, 2.5, 3, 4]
```

## 2.5.2. How can I remove an item from a list?

You can do it in a variety of ways.

- To begin, use the `remove()` method.
- You can also use the `pop()` method. If no index is supplied, the last item will be removed.
- The final way is to remove a specific item using the "`del`" keyword. If you want to scrap the entire list, you can use del.

```
my_list.remove(2.5)
print(my_list)
```

```
[1, 2, 3, 4]
```

```
my_list.pop(1)
print(my_list)
```

```
[1, 3, 4]
```

```
my_list = [0, 1, 2, 3, 4]
del my_list [3]
print(my_list)
```

```
[0, 1, 2, 4]
```

```
my_list = [0, 1, 2, 3, 4]
del my_list
```

## 2.5.3. Combine two lists into one

Use the + operator to combine two lists.

```
my_list = [0, 1, 2, 3, 4]
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]
print(my_list + second_list)
```

```
[0, 1, 2, 3, 4, 'a', 2, 'e', 4, 'i', 6, 'o', 8, 'u']
```

## 2.5.4. Change item value on your list

You can easily overwrite a value of one list items:

```
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]

second_list[0] = 1
second_list[2] = 3
second_list[4] = 5
second_list[6] = 7
second_list[8] = 9
print(second_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## 2.5.5. Loop through the list

Using for loop you can multiply the usage of certain items, similarly to what * operator does. Here's an example:

```
last_list = []
for x in range(1,4):
    last_list += ["Math"]
print(last_list)
```

```
['Math', 'Math', 'Math']
```

Here the `range(start, stop, step)` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

- start(Optional) An integer number specifying at which position to start. Default is 0
- stop(Required) An integer number specifying at which position to stop **(not included)**.
- step(Optional) An integer number specifying the incrementation. Default is 1

```
for x in range(1,4):
    print(x)
```

```
1
2
3
```

## 2.5.6. Copy a List

Use the built-in copy() function to replicate your data. Alternatively, you can copy a list with the list() method.

```
old_list = ["apple", "banana", "orange"]
new_list = old_list
print(new_list)
```

```
['apple', 'banana', 'orange']
```

```
old_list = ["apple", "banana", "orange"]
new_list = old_list.copy()
print(new_list)
```

```
['apple', 'banana', 'orange']
```

```
old_list = ["apple", "banana", "orange"]
new_list = list(old_list)
print(new_list)
```

```
['apple', 'banana', 'orange']
```

## 2.5.7. Lists Comprehensions

List comprehensions are a convenient way to make new lists from existing ones. You can also create with strings and tuples when using them.

Here an example how to create a new list with list comprehension.

```
list_variable = [x for x in iterable]
```

Here's a more complex example that features math operators, integers, and the `range()` function:

```
squared_evens = [x ** 2 for x in range(11) if x % 2 == 0]
print(squared_evens)
```

```
[0, 4, 16, 36, 64, 100]
```

# 2.6. Python Booleans

Booleans represent one of two values: `True` or `False`. It is useful to perform a filtering operation on a data.

In programming you often need to know if an expression is True or False. For example, when you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(8 > 2)
print(8 == 2)
print(2 < 8)
```

```
True
False
True
```

The next Python command prints a message based on whether the condition is `True` or `False`:

```
a = 22/7
b = 3.14

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

```
b is not greater than a
```

# 2.7. Python Dictionaries

Dictionaries are used to store key-value pairs. These key-value pairs offer a great way of organizing and storing data in Python. They are **mutable**, meaning you can change the stored information.

## 2.7.1. How to create a Python dictionary

Here's a quick example showcasing how to make an empty dictionary.

```
new_dict = {}
other_dict= dict()
```

When you want your values to be indexed by unique keys, they come in handy. Curly braces can be used to create a dictionary in Python. A colon is also used to separate a key and a value.

```
new_dict = {
    "brand": "Tesla",
    "model": "Model X",
    "year": 2021
}
print(new_dict)
```

```
{'brand': 'Tesla', 'model': 'Model X', 'year': 2021}
```

## 2.7.2. How to access a value in a dictionary

You can access any of the values in your dictionary the following way:

- dict["key"]

```
print(new_dict["model"])
```

```
Model X
```

## 2.7.3. Dictionary methods

- `keys()` method in Python Dictionary, returns a view object that displays a list of all the keys in the dictionary in order of insertion.
- `values()` method returns a view object that displays a list of all the values of the dictionary.
- `items()` method is used to return the list with all dictionary keys with values.

```
new_dict.keys()
```

```
dict_keys(['brand', 'model', 'year'])
```

```
new_dict.values()
```

```
dict_values(['Tesla', 'Model X', 2021])
```

```
new_dict.items()
```

```
dict_items([('brand', 'Tesla'), ('model', 'Model X'), ('year', 2021)])
```

Let's look at an example and see how lists compare to dictionaries.

Assume we have some movies and you want to keep track of their ratings. We also want to be able to quickly retrieve a movie's rating just knowing the title. We can accomplish this by employing two lists or a single dictionary. Take, for example, movies. The index("Ex Machina") code returns the index for the "Ex Machina" movie.

```
movies = ["Doctor Strage", "Venom", "Thor"]
ratings = [8.9, 7.8, 8.4]

movie_choice_index = movies.index("Venom")
print(ratings[movie_choice_index])
```

```
7.8
```

```
ratings = {
    "Doctor Strange": 8.9,
    "Venom": 7.8,
    "Thor" : 8.4
}

print(ratings["Venom"])
```

```
7.8
```

## 2.7.4. Operations with dictionaries

Our dictionaries allow us to add, edit, and delete information. We may simply use this code our `dict[key] = value` to add or update the data. When we wish to get rid of a key-value pair, we use `del(our dict[key])`.

```
ratings["Ant-Man"] = 8.3
print(ratings)

ratings["Doctor Strange"] = 9.1
print(ratings)

del(ratings["Thor"])
print(ratings)
```

```
{'Doctor Strange': 8.9, 'Venom': 7.8, 'Thor': 8.4, 'Ant-Man': 8.3}
```

```
{'Doctor Strange': 9.1, 'Venom': 7.8, 'Thor': 8.4, 'Ant-Man': 8.3}
{'Doctor Strange': 9.1, 'Venom': 7.8, 'Ant-Man': 8.3}
```

## 2.7.5. Loop Through the Dictionary

To implement looping, we can use `for` loop command.

```
ratings = {
    "Doctor Strange": 8.9,
    "Venom": 7.8,
    "Thor" : 8.4
}


#print all key names in the dictionary
for x in ratings:
    print(x)
```

```
Doctor Strange
Venom
Thor
```

```
#print all values in the dictionary
for x in ratings:
    print(ratings[x])
```

```
8.9
7.8
8.4
```

```
#loop through both keys and values
for x, y in ratings.items():
    print(x, y)
```

```
Doctor Strange 8.9
Venom 7.8
Thor 8.4
```

# 2.8. How to Define a Function

Python 3 allows you to define your own functions for your application in addition to using built-in functions. To summarize, a function is a set of programmed instructions that carry out a specific task. A function can be reused throughout your program once it has been properly defined, i.e. the same code can be reused.

A simple overview of how to define a function in Python can be found here:

Use `def` keyword followed by the function `name():`. The parentheses can contain any parameters that your function should take (or stay empty).

```
def name():
    print("What's your name?")
```

```
name()
```

```
What's your name?
```

```
# Define function with parameters
def product_info(productname, price):
    print("Product Name: " + productname)
    print("Price: " + str(price))

# Call function with parameters assigned as above
product_info("Apple Watch ",30000)
```

```
Product Name: Apple Watch
Price: 30000
```

```
# Call function with keyword arguments
product_info(productname= "Ipad Air 3", price=20000)
```

```
Product Name: Ipad Air 3
Price: 20000
```

# 2.9. If Statements (Conditional Statements) in Python

Similar to other programming languages, Python supports the basic logical conditions from math:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

You can leverage these conditions in various ways. But most likely, you'll use them in "if statements" and loops.

### 2.9.1. If Statement Example

The goal of a conditional statement is to check if it's True or False.

```
a = 22/7

if a > math.pi:
    print("a is greater than pi!")
```

```
a is greater than pi!
```

### 2.9.2. Elif Statements

The `elif` keyword instructs your program to try again, if the previous condition(s) were false.

The `else` keyword helps you add some additional filters to your condition clause.

Here's how an if-elif-else combo looks.

```
Score = 67

if Score >= 80:
    print("Your grade is A")
elif Score >= 75:
    print("Your grade is B+")
elif Score >= 70:
    print("Your grade is B")
elif Score >= 65:
    print("Your grade is C+")
elif Score >= 60:
    print("Your grade is C")
else:
    print("Your grade is D")
```

```
Your grade is C+
```

## 2.10. Loops in Python

**For loops** and **while loops** are two simple loop statements in Python that are useful to know.

Let's take a look at each one individually.

### 2.10.1. For Loop

For loop is a useful approach to iterate through a sequence such as a list, tuple, dictionary, string, and so on, as seen in the other sections of this Python basics.

An example of how to loop through a string is as follows:

```
for x in "Mahidol":
    print(x)
```

```
M
a
h
i

d
o
l
```

### 2.10.2. While Loops

A while loop allows you to run a group of statements as long as their condition is true.

```
#print as long as x is less than 8
i=1
while i< 8:
    print(i)
    i += 1
```

```
1
2
3
4
5
6
7
```

## 2.10.3. How to Break a Loop

Even if the condition is met, you can stop the loop from executing. Use the break statement in both while and for loops to accomplish this:

```
#print if x is less than 8, but skip four

i=1
while i< 8:
    print(i)
    if i == 4:
        break
    i += 1
```

```
1
2
3
4
```

```
print(i)
```

```
4
```

Previous
1. Introduction to Data Science

Next
3. Data Selection

By Pairote Satiracoo