# The Basics of Python Programming

# Chapter 1: Variables

Variables are spaces of a particular type that can store values like text, numbers, … anything. You can keep a reference to a variable by assigning it a name.

**Some examples (the text after a hash is a comment)**

```
1  x = 1        # variable with the name x and an integer value 1
2  y = “Jefke”  # variable with the name y and a string value “Jefke"
3  ltr = “d”    # variable with the name ltr and a character value “d”
4
5
```

**Numbers**

There are a couple of different types of numbers that variables can store. The most important once however are “Integers” and “Floating Point” values.

Integers are whole numbers and floating point numbers are not. For instance:
3 is an integer value and 12,33 is a floating point value.

```
1  a = 3       # Integer
2  b = 1.20    # Floating Point
3  c = 3,34    # Floating Point
4  d = 12      # Integer
5
```

**Characters**

Character values are simple. These are letters, spaces, commas, braces, etc. Basically anything that is not a number and is one character long. Examples:

```
1  a = “r”     # character r
2  b = “ “     # the space character
3  c = “_”     # an underscore character
4  d = “(“     # a left bracket character
5
```

**Strings**

A string is a text value. You can also see it as a collection of characters. Examples:

```
1  a = “Jefke”           # a string value “Jefke”
2  b = “Jantje“          # a string value “Jantje”
3  c = “Stoner”          # a string value “Stoner”
4  d = “Programming!“    # a string value “Programming!”
```

# Chapter 2: Collections

Collections are just that: collections of variables that are linked in some way. Variables are spaces that can store one value. But what if you had many similar values, like the names of the months in a year. Collections are a very good solution for this. There a few types.

**Tuples**

A tuple is a list of values like a list of names of your cats, a list of telephone numbers, etc. Once you define this list, it cannot be changed. Never. Like any programmer would say: “Tuples are immutable”. Some examples:

```
1  winter = (“Dec”, “Jan”, “Feb”, “Mar”)  # a tuple of strings
2  zip_codes = (2000, 2140, 3000)         # a tuple of integers
3  prices = (12,5, 2,4, 3,5, 7,99, 8,2)   # a tuple of floating point numbers
4
5
```

Each element can be accessed using an index (starting from 0). The first element has an index of 0, second an index of 1, etc.

```
1  winter = (“Dec”, “Jan”, “Feb”, “Mar”)  # a tuple of strings
2  winter[0]                              # a string value “Dec”
3  winter[2]                              # a string value “Feb”
4
5
```

**Lists**

A list is very similar to a tuple. It is also a list of values. The difference is that the values can be changed easily. Lists are mutable (modifiable, changeable, editable, you get the point).

```
1  cats = [“Jordy”, “Pluis”, “Grumpy”, “Tara”]  # a list of strings
2  winter[0]                                    # a string value “Jordy”
3  winter[2]                                    # a string value “Grumpy”
4  winter[1] = “Tom”                            # “Pluis” is changed to “Tom”
5  winter[1]                                    # a string value “Tom”
6  winter.append(“Speedy”)                      # add a string “Speedy” to the list
7  winter[4]                                    # a string “Speedy”
```

*! Notice the square brackets when defining a list.*

**Dictionaries**

What if you don't want to access elements using an index? What if you want to use a keyword? Dictionaries are coming to save you! A dictionary is a list of key-value pairs. Dictionaries are cool. Let's see what I mean.

```
1  pin_codes = {“Fortis”:1234, “KBC”:4455}      # a map called pin_codes
2  pin_codes[“KBC”]                             # an integer 4455
3  pin_codes[“ING”] = 2231                      # add a key-value pair “ING”:2231
4  pin_codes[“ING]                              # an integer 2231
5  del pin_codes[“Fortis”]                      # delete key-value pair “Fortis”:2134
6  pin_codes.has_key(“Fortis”)                  # False
7  pin_codes2 = {“Fortis”:2211, “Belfius”:1233} # a map called pin_codes2
8  pin_codes.update(pin_codes2)                 # update pin_codes using pin_codes2
9  pin_codes[“Belfius”]                         # an integer 1233
10 pin_codes[“Fortis”]                          # an integer 2211
11 new_codes = pin_codes.copy()                 # copy pin_codes into new_codes
12 new_codes[“Belfius”] = 9999                  # change 1233 in 9999
13 new_codes[“Belfius”]                         # an integer 9999
14 pin_codes[“Belfius”]                         # an integer 1233
```

You see how awesome they are? You can create, update, copy, delete, .. even sort.
Now you have headache, I know. Don't worry. It's gonna be fine.

# Chapter 3: Loops

Loops are structures you can use to some things over and over and over again. Imagine you want to print the numbers 0 to 10 on the screen. Programmers are lazy. We're not going to type print() 11 times. No way. We use loops to do the work. There are two kinds of loops.

**While Loop**

The while loop is very simple. It's going to loop (repeat over and over again) as long as some condition is true. When it's not true anymore, it stops.

```
1  number = 0                    # a variable "number" with value 0
2  while number < 4:             # will keep repeating as long as number < 4
3      print(number)             # prints the number to the screen
4      number = number + 1       # number is increased by 1
5
```

**For Loop**

The for loop is a little different. A for loop loops through each element in some collection. For example if we have a list with 3 elements, the for loop will execute for each of the elements. Let's see an example.

```
1  cats = ["Jordy", "Pluis", "Tom"]  # a list of strings
2  for cat in cats:                  # loops through cats and saves each element in cat
3      print(cat)                    # prints variable cat
4
5
```

# Chapter 4: Conditional Statements

Conditional statements are structures used for making choices. For example: you have a variable with a string value and you want to do different things for different values. Other languages have a couple of these structures but Python only has one.

**If .. elif .. else construct**

It works very simple. If some condition that is specified in if is true, then if executes. If another condition that is specified in elif is true, then elif executes. If none of these conditions are true, then else executes. Notice that elif and else are optional. They don't have to be specified.

```
1  message = “Hello”                    # a variable with value “Hello”
2  if message == “Hello”:               # if message equals to “Hello” then if executes
3      print(message)                   # prints variable message to the screen
4
```

```
1  message = “Yo yo!”                   # a variable with value “Yo yo!”
2  if message == “Yo!”:                 # if message equals “Yo!” => execute
3      print(message)                   # prints message
4  elif message == “Yo yo!”:            # if message equals “Yo yo!” => execute
5      print(message*2)                 # prints message two times
6  else:                                # if something else => execute
7      print(“Can’t print this!”)       # prints “Can’t print this!”
```

# Chapter 5: Functions

What are functions?!! Functions are pieces of code that do a specific task. You can think of them as little programs that do one thing. Functions can take input and they produce an output.

**Function definition**

First we need to define a function. Meaning: specify what input the function needs, what the function does and what output the function returns. Let's see.

```
1  def som(a, b):                # define function with name som and 2 parameters a, b
2      result = a + b            # a variable with value a + b
3      print(result)             # prints result to the screen
4
```

Here we've defined a function with name "som" and two input value (called parameters) a and b. The function adds the parameters together, saves that value to a variable named "result" and prints that variable to the screen. The function doesn't return anything. Let's create another function that does.

```
1  def som2(a, b):               # define function with name som and 2 parameters a, b
2      result = a + b            # a variable with value a + b
3      return result             # output of the function is result
4
```

These functions do essentially the same. they add two variables together. Only this time our function called "som2" doesn't print anything to the screen but return the result of the addition.

**The function call**

Let's call this function. Or in other words: let the computer execute it.
You can do that in two ways. You can call the function and not care about what it returns or you can call the function and save the return value in a variable. OK, so let's go.

```
1  som(1, 2):              # the function prints 3 on the screen
2  som2(1, 2)              # the function executes
3  a = som2(1,3)           # the function executes and the return value is saved in a
4  print(a)                # value 4 is printed on the screen
```

You see? This is an amazing tool for some lazy person (like a programmer). You write the function once and use it's functionality as many times you want.

# Chapter 6: Classes and Objects

Functions are cool for code re-use and all but they have a limitation. They don't store data as good as variables do. They are only the functionality part.

Sometimes you need more than that. Sometimes you have data (variables) that is very much related to the functionality. You want to make a car for example. A car has a model, a color, etc. This is the data part (variables). A car also has functionality (functions). It can drive, turn the lights on/off, make a signal sound, etc.

You can do that with functions and variables, but you will have to write a new function for each car you want to make. You will also need to create a separate set of variables for the details of each car. This is a lot of work. It's not good for lazy people. Frightening. Dangerous.

We need something else. Classes.

Classes are blueprints that combine data and functionality. You can create a lot of Objects from a blueprint, without changing it. Perfect. With one class you can create a lot of cars.

**Class definition**

Let's create a car class.

```
1  class Car:                        # define class with name Car (notice the capital C)
2      brand = "None"                # variable with string value "None"
3      model = "None"                # variable with string value "None"
4      color = "None"                # variable with string value "None"
5      lights = "off"                # variable with string value "off"
6      motor = "off"                 # variable with string value "off"
7
8      def start(self):              # define function start (notice "self")
9          self.motor = "on"         # value "off" changes to "on" (notice "self")
10
11     def drive(self):              # define function drive
12         if self.motor == "on":
13             print("driving..")
14         else:
15             print("can't drive. please start the car")
16
17     def toggle_lights(self):
18         if self.lights == "on":
19             self.lights = "off"
20         elif self.lights == "off":
21             self.lights = "on"
22
23     def are_lights_on(self):
24         if self.lights == "on":
25             return True
26         else:
27             return False
```

It's pretty clear. We define a class with name Car, create a couple of variables and define a few functions. There is not much new here but there are a couple of weird things. The way we define functions is a little different (what is that self thing??!!!) and the way we access variables is different (that stupid self again!!!).

Well, that keyword self is used to let python know that you define a function that belongs to this object, not a global function. This way, the function cannot be accessed without an object, created using this class. The same thing happens with variables. When you access a variable x for example, you have to choices. You can access a global variable with the name x that is created outside the class. Or you can access the variable that is created inside the class.
When you do the last, you need to specify self.

It may be confusing at first. When you get used to it, it's actually very easy.

**Using a Class**

There is one way you can use the class. Just like there is one way to use a blueprint. It's used to create Objects (sometimes called instances). Let's do it.

```
1  car1 = Car()              # create a car
2  car1.brand = "BMW"        # specify a brand
3  car1.model = "X3"         # specify a model
4  car1.color = "black"      # specify a color
5  car1.start()              # start the car
6  car1.drive()              # drive the car
7  lights = car1.are_lights_on()
8  if not(lights):
9      car1.toggle_lights()
```

Now that is cool. In the same way we can create a lot of cars. We specify the data we want and we have a lot of different cars without changing the code. We reached the ultimate lazy. I love it.

**The __init__ method**

There is one more thing to know. There is a special method (functions that are defined in classes are called methods) called __init__() that is called automatically when you create an object using a class. It is called "The constructor".

It is a very handy tool because there we can specify what happens when an object is created. Take our Car class for example. In our previous implementation we needed to set a few properties of a car (variables inside classes are called properties) after we created the car. Wouldn't it be nice if we could do that when created the object, not after? We can use our __init__ method. We can let the method take some parameters that will set the needed properties for us. Let's try that.

```
1  class Car:                       # define class with name Car (notice the capital C)
2      brand = "None"               # variable with string value "None"
3      model = "None"               # variable with string value "None"
4      color = "None"               # variable with string value "None"
5      lights = "off"               # variable with string value "off"
6      motor = "off"                # variable with string value "off"
7
8      def __init__(self, pbrand, pmodel, pcolor):
9          self.brand = pbrand
10         self.model = pmodel
11         self.color = pcolor
12
13     def start(self):             # The implementation of our methods..
14     def drive(self):             # Not shown here to safe space
15     def toggle_lights(self):
16     def are_lights_on(self):
17
18 car2 = Car("Volvo", "V40", "blue")  # This is so simple, right? :-)
19 car2.drive()
```

# Chapter 7: Exercises

Let's make some rules. Each exercise will be given a name, a description and some test cases. Your job is to write a correct implementation of a function or a class and save it in a new file. The name of the file should be the same as the name of the exercise (Use underscores instead of spaces to separate words!!!). All files should be committed to GitHub in the project Python_Exercises, in the master branch.

**Exercise 1:**

Given an int n, write a function difference() that returns the absolute difference between n and 20. When n is bigger than 20, return double the difference.

difference(19) —> 1
difference(10) —> 10
difference(23) —> 6

**Exercise 2:**

Given an int n, write a function near() that returns True if n is near 10 from 100 or 200 and otherwise returns False. Also return that difference.
Tip: use commas to return more than one value. Like so: `return a, b`

near(89) —> False, 11
near(103) —> True, 3
near(195) —> True, 5

**Exercise 3:**

Given a non-empty string str and an int n, write a function missing() that returns a new string where a character at index n is missing.

missing("hello", 1) —> "hllo"
missing("water", 3) —> "watr"
missing("kat", 0) —> "at"

**Exercise 4:**

You have two monkeys, jantje and jefke. Write a function trouble() that takes jantje_smiling and jefke_smiling as parameters and returns True is they are both smiling or none are smiling. Otherwise return False.

trouble(True, True) —> True
trouble(False, True) —> False
trouble(False, False) —> True

**Exercise 5:**

You are driving too fast and the police stops you. Write a function speed() that calculates which ticket you get. 0 = no ticket, 1 = small ticket and 2 = big ticket. You get no ticket if your speed is below 60 (inclusive), a small ticket if your speed is between 61 and 80 (inclusive) and a big ticket when the speed is above 80. If it's your birthday, your speed can be 5 higher in all cases. The function takes 2 parameters: int speed and boolean birthday.

speed(61, False) —> 1
speed(64, True) —> 0
speed(80, False) —> 2

**Exercise 6:**

Given two ints, a and b, write a function weird_sum() that returns their sum. However, sums between 10 and 15 (inclusive) are forbidden. In that case, return 20.

weird_sum(3,4) —> 7
weird_sum(5,7) —> 20
weird_sum(4,11) —> 20

**Exercise 7:**

Given an int day indicating a day of the week (0=monday, 1=tuesday, …, 6=sunday) and a boolean vacation indicating if we are on vacation. Write a function alarm() that returns a string in the form of "7:00" indicating when the alarm goes off. On a weekday the alarm should ring at "7:30". In the weekend at "10:00". If we are on vacation the alarm ring at "10:00" on a weekday and is "off" in the weekend.

alarm(1, False) —> "7:30"
alarm(3, True) —> "10:00"
alarm(5, True) —> "off"
alarm(6, False) —> "10:00"

**Exercise 8:**

Given an array of strings. Write a function part() that returns the first three letters of each string, separated by commas.

part(["hello", "train", "mushroom"]) —> "hell, trai, mush"
part(["cats", "leave", "dark"]) —> "cat, lea, dar"