

# Python Variables and Data Types

## Variables

One of the main concepts in programming is **variables**. They are your best friends. You will deal with them all the time. You will use them to store information. They will represent your data input.

```
In [1]: x = 5  
  
In [2]: x  
Out[2]: 5
```

Let's say you want to have a variable x that is equal to the value of 5 and then ask the computer to tell you the value of that variable. Type x equals 5. Write x and then execute. And here's the result – 5.

```
In [3]: y = 8  
  
In [5]: print (y)  
8
```

An alternative way to execute the instruction that will provide the value we assigned to y would be to use the print command.

If we say "print(y)", the machine will simply execute this command and provide the value of y *as a statement*.

## Numbers and Boolean Values

When programming, not only in Python, if you say that a variable has a numeric value, you are being ambiguous. The reason is that numbers can be **integers** or floating points, also called **floats**, for instance.

Term	Definition
<b>Integer</b>	Positive or negative whole numbers without a decimal point Example: 5, 10, -3, -15
<b>Floating point (float)</b>	Real numbers. Hence, they have a decimal point Example: 4.75, -5.50, 11.0
<b>Boolean value</b>	a True or False value, corresponding to the machine's logic of understanding 1s and 0s, on or off, right or wrong, true or false. Example: True, False

## Strings

Strings are text values composed of a sequence of characters.

```
In [2]: 'George'
Out[2]: 'George'

In [3]: "George"
Out[3]: 'George'

In [4]: print ('George')
George

In [5]: print ("George")
George
```

Type single or double quotation marks around the name George. Python displays 'George' if you don't use the print command. Should you use print, the output will be shown with no quotes – you'll be able to see plain text.

Strings are text values composed of a sequence of characters.

```
In [11]: "I'm fine"
Out[11]: "I'm fine"

In [12]: 'I\'m fine'
Out[12]: "I'm fine"

In [13]: 'Press "Enter"'
Out[13]: 'Press "Enter"'
```

To type “I’m fine” you’ll need the apostrophe in the English syntax, not for the Pythonic one. In such situations, you can distinguish between the two symbols – put the text within double quotes and leave the apostrophe, which technically coincides with the single quote between I and M.

An alternative way to do that would be to leave the quotes on the sides and place a back slash before the apostrophe within the phrase. This backslash is called an **escape character**, as it changes the interpretation of characters immediately after it.

To state “press “Enter””, the outer symbols must differ from the inner ones. Put single quotes on the sides.

Strings are text values composed of a sequence of characters.

```
In [17]: print ('Red ' + 'car')
```

```
Red car
```

```
In [18]: print ('Red', 'car')
```

```
Red car
```

Say you wish to print “Red car” on the same line. “Add” one of the strings to the other by typing in a plus sign between the two. Put a blank space before the second apostrophe of the first word.

(In [17])

Type “print ‘Red’”, and then put a comma, which is called a **trailing comma**, and Python will print the next word, ‘car’, on the same line, separating the two words with a blank space. (In [18])

## Basic Python Syntax

### Arithmetic Operators

```
In [1]: 1 + 2
Out[1]: 3
```

In the equation you see here, 1 and 2 are called **operands**, The plus and minus signs are called **operators**, and given they also represent arithmetic operations, they can be called **arithmetic operators**.

Operator	Description
+	Addition
-	Subtraction
/	Division <i>Note: If you want to divide 16 by 3, when you use Python 2, you should look for the quotient of the float 16 divided by 3 and not of the integer 16 divided by 3. So, you should either transform the number into a float or type it as a float directly.</i>
%	Returns remainder
*	Multiplication
**	Performs power calculation

## The Double Equality Sign

```
In [1]: y = 5 ** 3
```

```
In [2]: y
```

```
Out[2]: 125
```

```
In [3]: y == 125
```

```
Out[3]: True
```

```
In [4]: y == 126
```

```
Out[4]: False
```

The equals sign when programming means “assign” or “bind to”. For instance, “assign 5 to the power of 3 to the variable y”; “bind 5 to the power of 3 to y”. From that moment for the computer, y will be equal to 125.

When you mean equality between values and not assignment of values in Python, you’ll need **the double equality sign**. Anytime you use it, you will obtain one of the two possible outcomes – “True” or “False”.

## Reassign Values

```
In [1]: z = 1  
z
```

```
Out[1]: 1
```

```
In [2]: z = 3  
z
```

```
Out[2]: 3
```

```
In [3]: z + 5
```

```
Out[3]: 8
```

```
In [4]: z = 7  
z
```

```
Out[4]: 7
```

If you assign the value of 1 to a variable z, your output after executing z will be 1. After that, if you assign 3 to the same variable z, z will be equal to 3, not 1 anymore.

Python **reassigns** values to its objects. Therefore, remember the last command is valid, and older commands are overwritten.

## Add Comments

```
In [2]: # This is just a comment and not code!  
print (7,2)
```

7 2

```
In [3]: # Comment 1  
# Comment 2  
print (300)
```

300

**Comments** are sentences not executed by the computer; it doesn't read them as instructions. The trick is to put a hash sign at the beginning of each line you would like to insert as a comment.

If you would like to leave a comment on two lines, don't forget to place the hash sign at the beginning of each line.

## Line Continuation

```
In [ ]: 2.0 * 1.5 + \
```

5

You might prefer to send part of the code to the next line. So, 2.0 times 1.5 plus 5 could be written in two lines, and the machine could still read it as one command. This could be achieved by putting a back slash where you would like the end of the first line to be. It indicates you will continue the same command on a new line.

## Indexing Elements

```
In [ ]: "Friday"[]
```

`"Name_of_variable"[index_of_element]`

Is it possible to extract the letter “d”?

Yes, you can do that by using square brackets. And within them, you should specify the position of the letter we would like to be extracted.

*Note:*

*Make sure you don't mistake brackets for parentheses or braces:*

*parentheses - ()*

*brackets - []*

*braces- {}!*

The image shows a Python IDE interface. The top part has a code editor with the following content:

```
In [1]: "Friday"[3]  
Out[1]: 'd'
```

Below the code editor is a Python logo and a text box that says: "Count from 0, not from 1!".

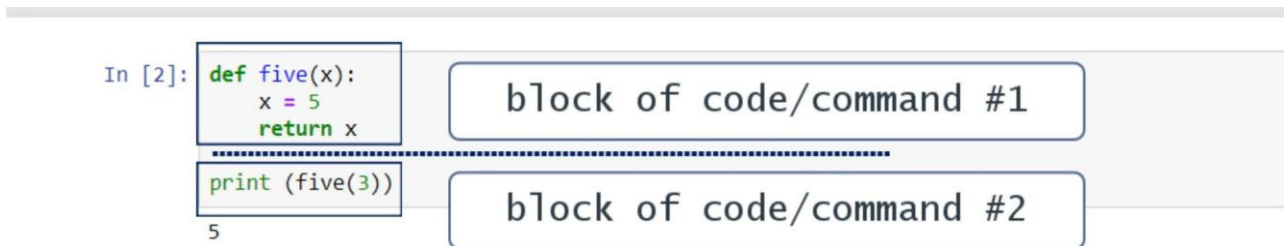
Below the text box is a visual representation of the string "Friday" as a sequence of characters in a dark blue bar: F, R, I, D, A, Y. Below each character is a white box containing its index: 0, 1, 2, 3, 4, 5. The index 3 box is circled in yellow, and the letter 'D' is underlined with a blue double line.

A very important thing you should remember is that, **in Python, we count from 0, not from 1!** 0, 1, 2, 3, 4, and so on. That's why I'll ask for the 4th letter, 'd', by writing 3 here.



## Structure Your Code with Indentation

The way you apply indentation in practice is important, as this will be the only way to communicate your ideas to the machine properly.



Def and Print form two separate and, written in this way, clearly distinguishable **blocks of code** or **blocks of commands**.

Everything that regards the function is written with one indentation to the inside. Once you decide to code something else, start on a new line with no indentation. The blocks of code are more visible, and this clarifies the logic you are applying to solve your problem.

## 4.8 Comparison Operators

Operator	Description
<code>==</code>	Verifies the left and right side of an equality are equal
<code>!=</code>	Verifies the left and right side of an equality are not equal
<code>&gt;</code>	Greater than
<code>&lt;</code>	Less than
<code>&gt;=</code>	Greater than or equal to
<code>&lt;=</code>	Less than or equal to

## Logical Operators

Briefly, the **logical operators** in Python are the words **"NOT"**, **"AND"**, and **"OR"**. They compare a certain amount of statements and return Boolean values – "True" or "False" – hence their second name, **Boolean operators**.

Operator	Description
<b>not</b>	Leads to the opposite of the given statement Example: "not True" leads to False
<b>and</b>	Checks whether the two statements around it are "True" Example: "True and False" leads to False
<b>or</b>	Checks whether at least one of the two statements is "True" Example: "False or True" leads to True

You must respect the **order of importance** of these three operators. It is: "not" comes first, then we have "and", and finally "or".

## Identity Operators

The **identity operators** are the words **"is"** and **"is not"**. They function similar to the double equality sign and the exclamation mark and equality sign we saw earlier.

```
In [1]: 5 is 6
```

```
Out[1]: False
```

```
In [2]: 5 == 6
```

```
Out[2]: False
```

```
In [3]: 5 is not 6
```

```
Out[3]: True
```

```
In [4]: 5 != 6
```

```
Out[4]: True
```

# Conditional Statements

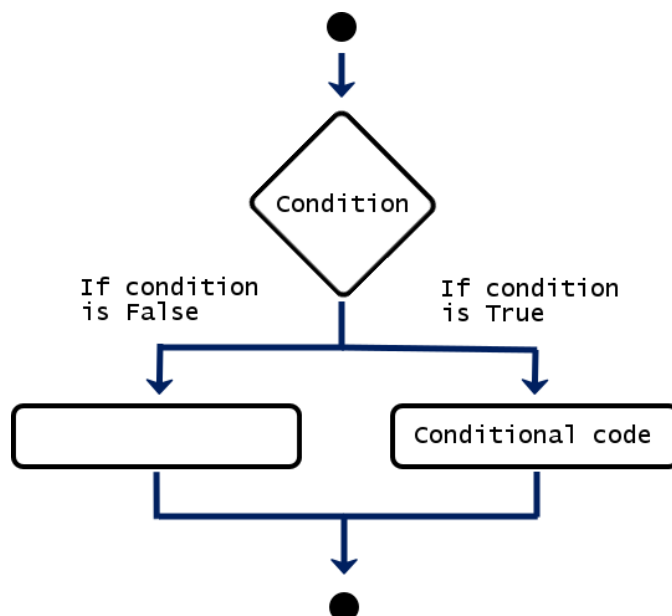
## Introduction to the IF statement

A prominent example of **conditional statements** in Python is the **"IF" statement**. What you can use it for is intuitive, but it is very important to learn the syntax.

```
In [1]: if 5 == 15 / 3:  
        print ("Hooray!")
```

Hooray!

```
In [ ] if condition:  
        conditional code
```



The graph could help you imagine the process of the conditionals. Before it displays the outcome of the operation, the machine follows these logical steps. If the conditional code is not to be executed because the if-condition is not true, the program will directly lead you to some other output or, as it is in this case, to nothing. After any of the two situations, the machine will go to the next black point and will progress from there on.

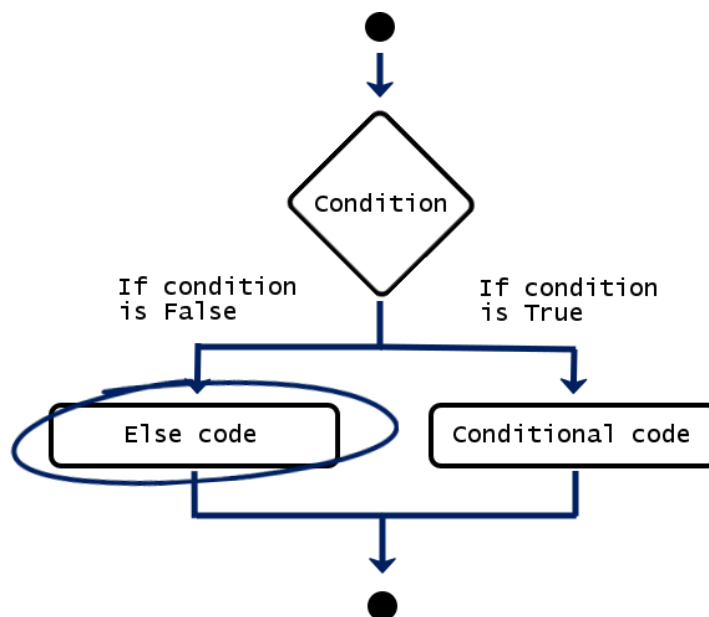
## Add an ELSE statement

"ELSE" will tell the computer to execute the successive command in all other cases.

```
In [1]: x = 1  
  
if x > 3:  
    print ("Case 1")  
else:  
    print ("Case 2")
```

Case 2

```
if condition:  
    conditional code  
else:  
    else code
```



Instead of leading to no output, if the condition is false, we will get to an else code. Regardless whether the initial condition is satisfied, we will get to the end point, so the computer has concluded the entire operation and is ready to execute a new one.

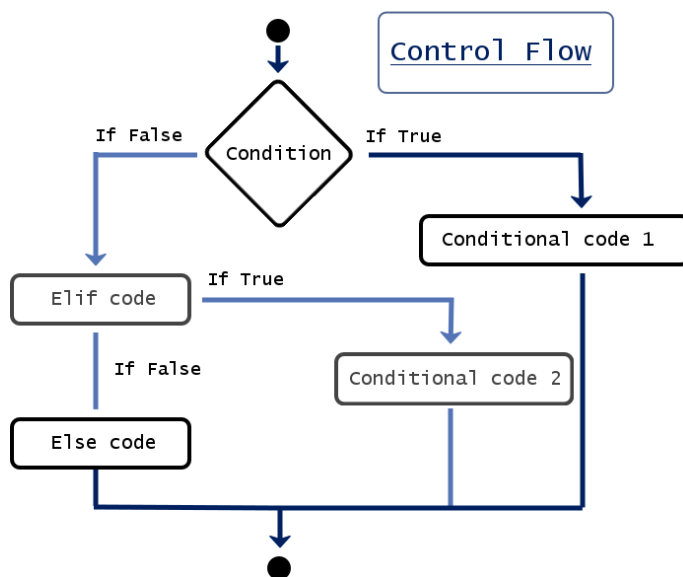
## Else if, for Brief - ELIF

If y is not greater than 5, the computer will think: “else if y is less than 5”, written “**elif** y is less than 5”, then I will print out “Less”.

```
In [1]: def compare_to_five(y):  
        if y > 5:  
            return "Greater"  
        elif y < 5:  
            return "Less"  
        else:  
            return "Equal"
```



Know that you can add as many elif statements as you need.

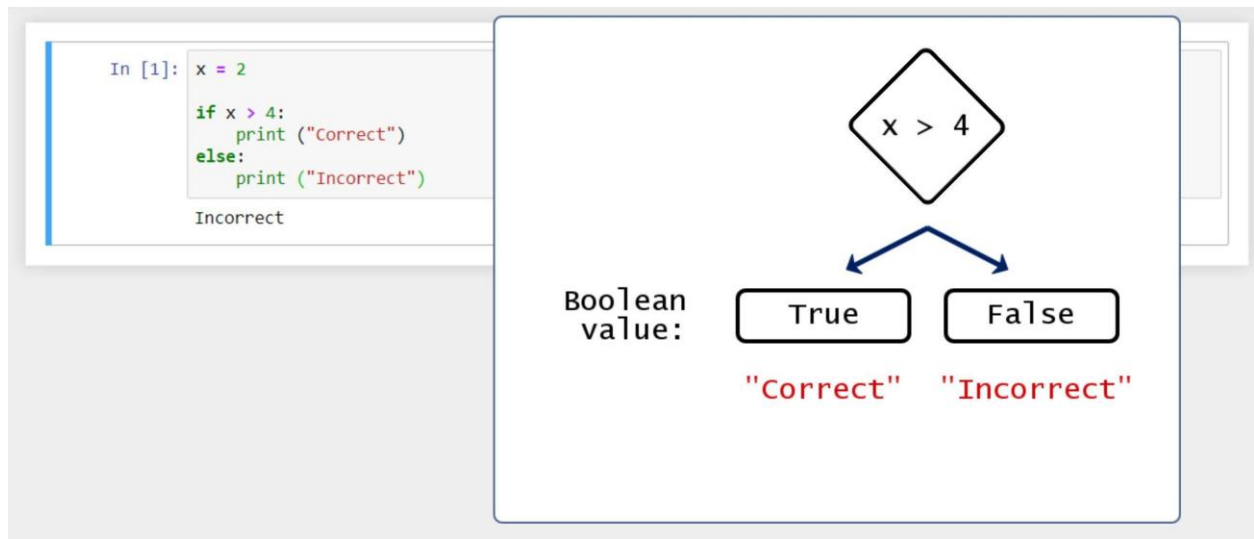


A very important detail you should try to remember is the computer always reads your commands from top to bottom. Regardless of the speed at which it works, it executes only one command at a time. Scientifically speaking, the instructions we give to the machine are part of a control flow. This is something like the flow of the logical thought of the computer, the way the computer thinks – step by step, executing the steps in a rigid order.

When it works with a conditional statement, the computer’s task will be to execute a specific command once a certain condition has been satisfied. It will read your commands from the if- statement at the top, through the elif-statements in the middle, to the else- statement at the end. The first moment the machine finds a satisfied condition, it will print the respective output and will execute no other part of the code from this conditional.

## A Note on Boolean Values

From a certain perspective, everything in a computer system is Boolean, comprising sequences of 0s and 1s, “False” and “True”. This is why we are paying attention to the Boolean value. It helps us understand general computational logic and the way conditionals work in Python.

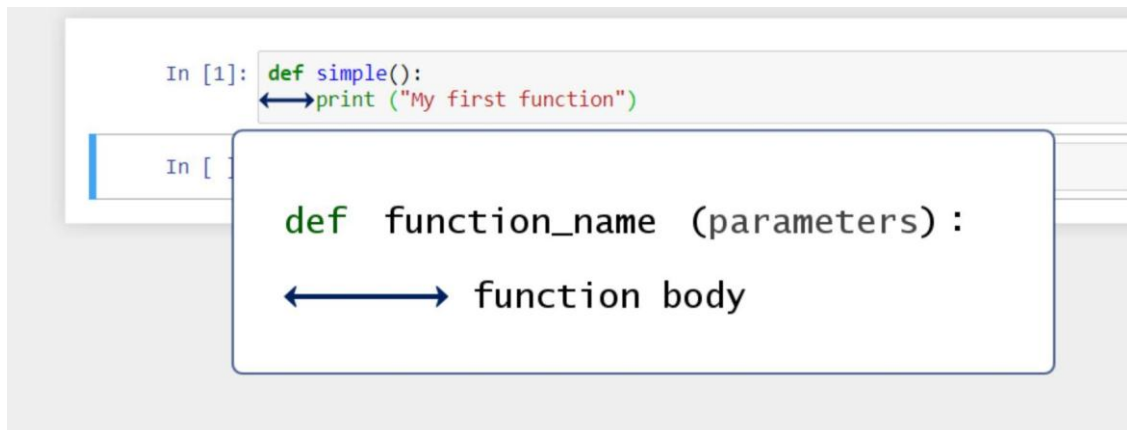


Basically, after you insert your if-statement, the computer will attach a Boolean value to it. Depending on the value of its outcome, “True” or “False”, it will produce one of the suggested outputs, “Correct” or “Incorrect”.

# Functions

## Defining a Function in Python

*Python's **functions** are an invaluable tool for programmers.*



To tell the computer you are about to create a function, just write **def** at the beginning of the line. Def is neither a command nor a function. It is a **keyword**. To indicate this, Jupyter will automatically change its font color to green. Then, you can type the **name of the function** you will use. Then you can add a pair of **parentheses**. Technically, within these parentheses, you could place the **parameters** of the function if it requires you to have any. It is no problem to have a function with zero parameters.

To proceed, don't miss to put a colon after the name of the function.

Since it is inconvenient to continue on the same line when the function becomes longer, it is much better to build the habit of laying the instructions on a new line, with an **indent** again. Good legibility counts for a good style of coding!

## Creating a Function with a Parameter

```
In [3]: def plus_ten(a):  
        return a + 10
```

```
In [4]: plus_ten(2)
```

```
Out[4]: 12
```

```
In [5]: plus_ten(5)
```

```
Out[5]: 15
```

```
In [ ]:
```

```
def function_name (parameters):
```

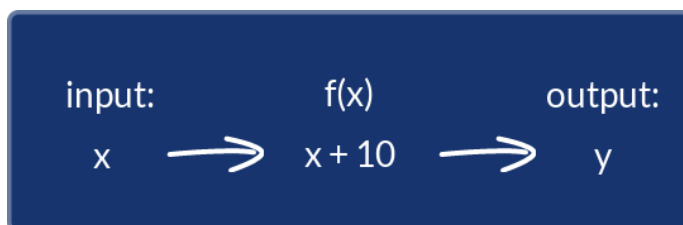
```
    but call
```

```
    plus_ten (arguments):
```

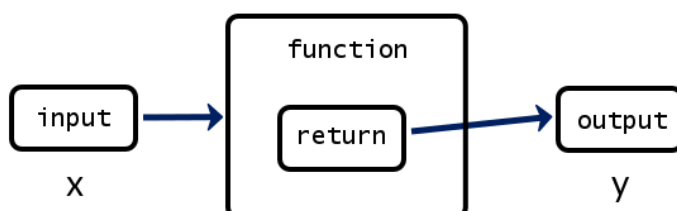
Don't forget to **return** a value from the function. We will need `plus_ten(a)` to do a specific calculation for us and not just print something.

Pay attention to the following. When we define a function, we specify in parentheses a **parameter**. In the `plus_ten()` function, "a" is a parameter. Later, when we call this function, it is correct to say we provide an argument, not a parameter. So we can say "call `plus_ten()` with an argument of 2, call `plus_ten()` with an argument of 5".

In programming, **return** regards the value of  $y$ ; it just says to the machine "after the operations executed by the function  $f$ , return to me the value of  $y$ ". "Return" plays a connection between the second and the third step of the process. In other words, *a function can take an input of one or more variables and return a **single** output composed of one or more values*. This is why "return" can be used only once in a function.



*People often confuse print and return, and the type of situations when we can apply them.*



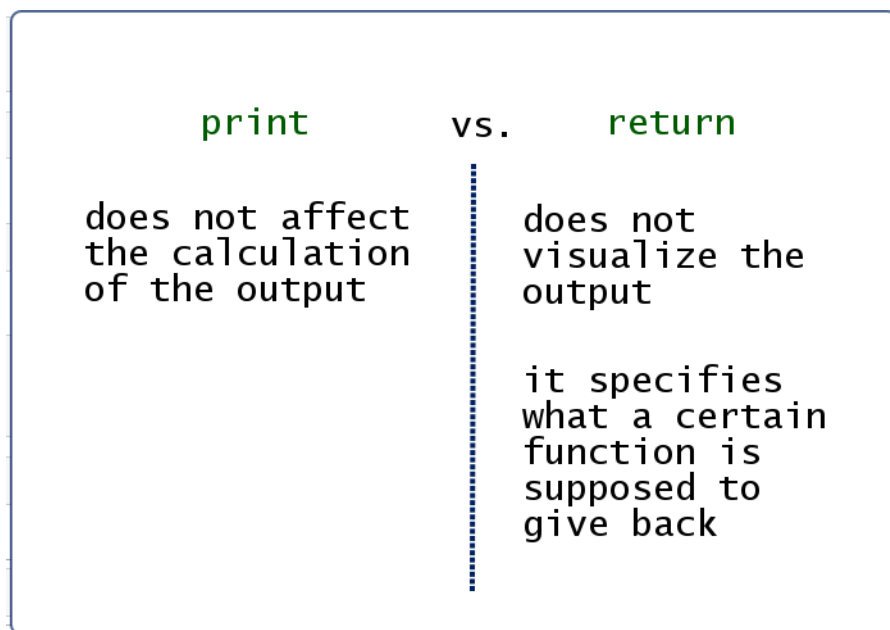


## print vs. return

**print()** takes a statement or, better, an object, and provides its printed representation in the output cell. It just makes a certain statement visible to the programmer. Otherwise, print does not affect the calculation of the output.

Differently, **return** does not visualize the output. It specifies what a certain function is supposed to give back.

It's important you understand what each of the two keywords does. This will help you a great deal when working with functions.



## Using a Function in Another Function

It isn't a secret we can have a function within the function.


In `with_bonus(w_hours)`, you can return directly the wage with working hours as an output, which would be the value obtained after the wage function has been run, plus 50.

```
In [ ]: def wage(w_hours):  
        return w_hours * 25  
  
        def with_bonus(w_hours):  
            return wage(w_hours) + 50
```

## Creating Functions Containing a Few Arguments

You can work with more than one parameter in a function. The way this is done in Python is by enlisting all the arguments within the parentheses, separated by a comma.

```
In [1]: def subtract_bc(a,b,c):  
        result = a - b*c  
        print ('Parameter a equals', a)  
        print ('Parameter b equals', b)  
        print ('Parameter c equals', c)  
        return result
```



```
def function_name (parameter #1, parameter #2, ... ):
```

You can call the function for, say, 10, 3, and 2. You will get 4. Just be careful with the order in which you state the values. In this case, we assigned 10 to the variable a, 3 to b, and 2 to c.

```
In [2]: subtract_bc(10,3,2)  
        Parameter a equals 10  
        Parameter b equals 3  
        Parameter c equals 2  
  
Out[2]: 4
```

```
In [ ]:
```

Otherwise, the order won't matter if and only if you specify the names of the variables within the parentheses.

```
In [3]: subtract_bc(b=3,a=10,c=2)  
        Parameter a equals 10  
        Parameter b equals 3  
        Parameter c equals 2  
  
Out[3]: 4
```

## Notable Built-In Functions in Python

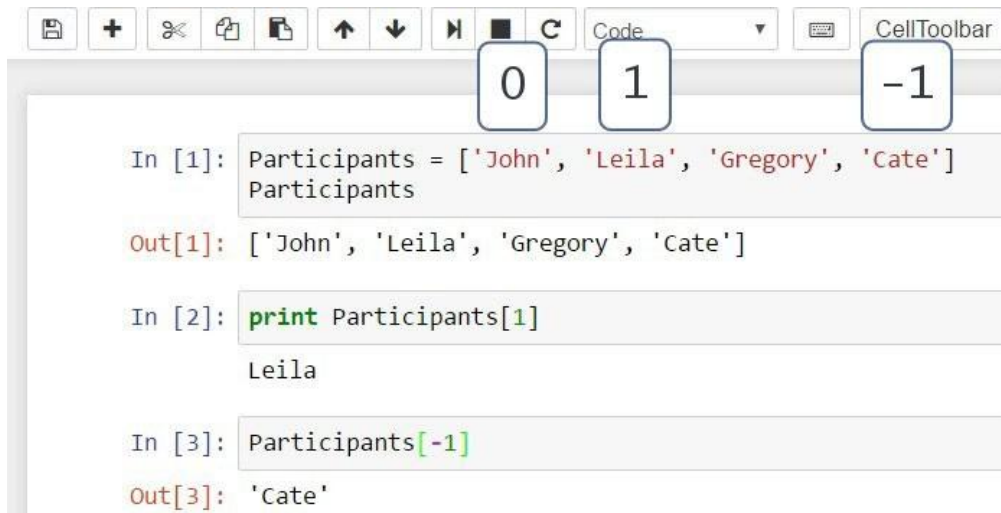
When you install Python on your computer, you are also installing some of its **built-in functions**. This means you won't need to type their code every time you use them – these functions are already on your computer and can be applied directly.

Function	Description
<b>type()</b>	obtains the type of variable you use as an argument
<b>int()</b>	transforms its argument in an integer data type
<b>float()</b>	transforms its argument in a float data type
<b>str()</b>	transforms its argument in a string data type
<b>max()</b>	Returns the highest value from a sequence of numbers
<b>min()</b>	Returns the lowest value from a sequence of numbers
<b>abs()</b>	Allows you to obtain the absolute value of its argument
<b>sum()</b>	Calculates the sum of all the elements in a list designated as an argument
<b>round(x,y)</b>	returns the float of its argument (x), rounded to a specified number of digits (y) after the decimal point
<b>pow(x,y)</b>	returns x to the power of y
<b>len()</b>	returns the number of elements in an object

# Sequences

## Lists

A **list** is a type of sequence of data points such as floats, integers, or strings.



The screenshot shows a Jupyter Notebook interface with a toolbar at the top containing icons for save, add, delete, copy, paste, undo, redo, and a 'Code' dropdown menu. Below the toolbar, three index values are highlighted in blue boxes: '0', '1', and '-1'. The notebook contains three input-output pairs:

```
In [1]: Participants = ['John', 'Leila', 'Gregory', 'Cate']
Participants
Out[1]: ['John', 'Leila', 'Gregory', 'Cate']

In [2]: print Participants[1]
Leila

In [3]: Participants[-1]
Out[3]: 'Cate'
```

You can access the Participants list by indexing the value 1. This means you have extracted the second of the elements in this list variable ['Leila'].

In addition, there is a way to get to the last element from your list – start counting from the end towards the beginning. Then, you'd need the minus sign before the digit and don't fall in the trap of thinking we begin enumerating from 0 again! To obtain "Cate", you have to write -1.

## Help Yourself with Methods

Here is the syntax that allows you to call ready-made **built-in methods** that you do not have to create on your own and can be used in Python directly.

After the name of the **object**, which in this case is the “Participants” list, you must put a dot called a **dot operator**. The dot operator allows you to **call** on or **invoke** a certain method. To call the method “append”, state its name, followed by **parentheses**.

object . method ()

```
In [8]: Participants.append("Dwayne")  
Participants
```

```
Out[8]: ['John', 'Leila', 'Maria', 'Dwayne']
```

To insert the name “Dwayne” in our list, you must put the string “Dwayne” in inverted commas between the parentheses.

```
In [9]: Participants.extend(['George', 'Catherine'])  
Participants
```

```
Out[9]: ['John', 'Leila', 'Maria', 'Dwayne', 'George', 'Catherine']
```

Alternatively, the same result can be achieved by using the .extend() method. This time, within the parentheses, you’ll have to add brackets, as you are going to extend the Participants list by adding a list specified precisely in these parentheses.

## List Slicing

Many of the problems that must be solved will regard a tiny portion of the data, and in such cases, you can apply slicing.

```
In [1]: Participants = ['John', 'Leila', 'Maria', 'Dwayne', 'George', 'Catherine']

In [2]: Participants
Out[2]: ['John', 'Leila', 'Maria', 'Dwayne', 'George', 'Catherine']

In [3]: Participants[1:3]
Out[3]: ['Leila', 'Maria']

In [ ]:
```

Imagine you want to use the “Participants” list to obtain a second much smaller list that contains only two names –Leila and Maria. In Pythonic, that would mean to extract the elements from the first and second position. To access these elements, we will open square brackets, just as we did with indexing, and write 1 colon 3. The first number corresponds precisely to the first position of interest, while the second number is one position above the last position we need.

## Tuples

**Tuples** are another type of data sequences, but differently to lists, they are *immutable*. Tuples cannot be changed or modified; you cannot append or delete elements.

```
In [1]: x = (40,41,42)
        x
Out[1]: (40, 41, 42)

In [2]: y = 50,51,52
        y
Out[2]: (50, 51, 52)

In [3]: a,b,c = 1,4,6
        c
Out[3]: 6
```

The syntax that indicates you are having a tuple and not a list is that the tuple's elements are placed within parentheses and not brackets.

The tuple is the default sequence type in Python, so if you enlist three values here, the computer will perceive the new variable as a tuple. We could also say the three values will be **packed** into a tuple.

For the same reason, you can assign a number of values to the same number of variables. On the left side of the equality sign, add a tuple of variables, and on the right, a tuple of values. That's why the relevant technical term for this activity is **tuple assignment**.

## Dictionaries

**Dictionaries** represent another way of storing data.

```
In [1]: dict = {'k1': "cat", 'k2': "dog", 'k3': "mouse", 'k4': "fish"}  
dict
```

```
Out[1]: {'k1': 'cat', 'k2': 'dog', 'k3': 'mouse', 'k4': 'fish'}
```

```
In [2]: dict['k1']
```

```
Out[2]: 'cat'
```

Each value is associated with a certain **key**. More precisely, a key and its respective value form a **key-value pair**.

After a certain dictionary has been created, a value can be accessed by its key, instead of its index!

```
In [ ]: dict['k5'] = 'parrot'
```

```
dictionary_name [new_key_name] = new_value_name
```

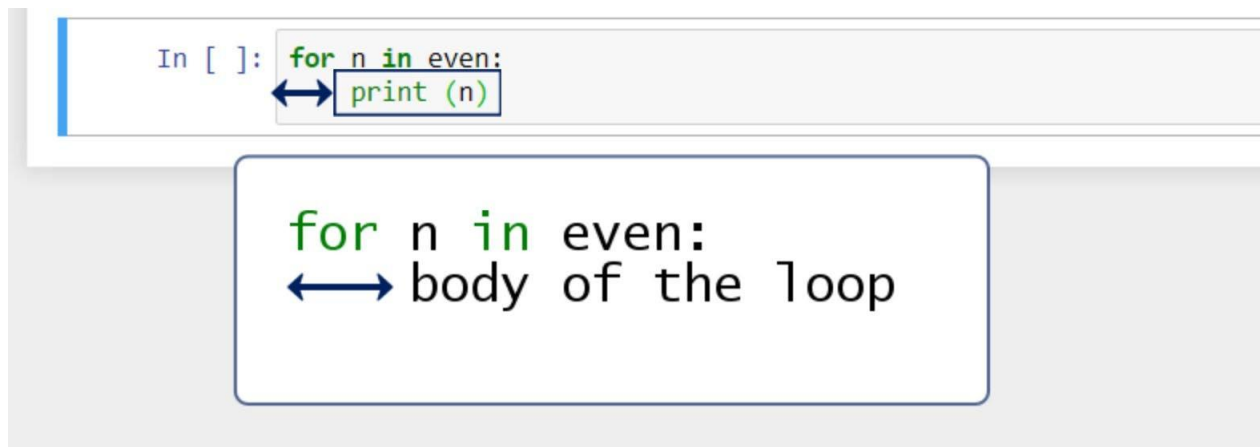
Similarly, as we could do with lists, we can add a new value to the dictionary in the following way: the structure to apply here is dictionary name, new key name within brackets, equality sign, and the name of the new value.



## Iteration

### For Loops

**Iteration** is a fundamental building block of all programs. It is the ability to execute a certain code repeatedly.



The list “even” contains all the even numbers from 0 to 20. “for n in even”, colon, which would mean for *every element* n in the list “even”, do the following: print that element.

The command in the loop body is performed once *for each* element in the even list.

## While Loops and Incrementing

The same output we obtained in the previous lesson could be achieved after using a while loop, instead of a for loop. However, the structure we will use will be slightly different.

```
In [1]: x = 0
        while x <= 20:
            print (x, end = " ")
            x = x + 2

0 2 4 6 8 10 12 14 16 18 20
```

Initially, we will set a variable `x` equal to zero. And we'll say: **while** this value is smaller than or equal to 20, print `x`.

We want to get the loop to end. What is supposed to succeed, the loop body in the "while" block, is a line of code that specifies a change in `x` or what has to happen to `x` after it is printed. In our case, we will tell the computer to bind `x` to a value equal to `x + 2`.

```
In [2]: x = 0
        while x <= 20:
            print (x, end = " ")
            x += 2

0 2 4 6 8 10 12 14 16 18 20
```

In programming terms, adding the same number on top of an existing variable during a loop is called **incrementing**. The amount being progressively added is called an **increment**. In our case, we have an increment of 2.

The Pythonic syntax offers a special way to indicate incrementing: **`x += 2`**

## Create Lists with the range() Function

When you need to randomize data points and lists with data points, you can use Python's built-in range function.

The syntax of the function is the following:

```
           0           1
range(start, stop, step)

start = the first number in the list
stop = the last value +1
step = the distance between each two consecutive values
```

The stop value is a required input, while the start and step values are optional. If not provided, the start value will be automatically replaced with a 0, and the step value would be assumed to be equal to 1.

```
In [1]: range(10)
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: range(3,7)
Out[2]: [3, 4, 5, 6]

In [3]: range(1,20,2)
Out[3]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

*range(10)* will provide a list of 10 elements, starting from 0, implied after not indicating a start value, and ending at the tenth consecutive number – 9. In another cell, if in the “range” function we declare as arguments 3 and 7, for instance, Python will accept 3 as a start value, and 7 as a stop value of the range. So, we’ll have 4 elements – 3, 4, 5, and 6.

To specify a step value in a range, the other two arguments must be chosen as well. *range(1,20,2)* creates a list with all the odd numbers from 1 to 19 included. It will start with the number 1, and the list will end with number 19 (which equals the stop value 20 minus 1), stating only the odd numbers.

## Use Conditional Statements and Loops Together

```
In [2]: for x in range(20):  
        if x % 2 == 0:  
            print (x, end = " ")  
        else:  
            print ("Odd", end = " ")
```

0 Odd 2 Odd 4 Odd 6 Odd 8 Odd 10 Odd 12 Odd 14 Odd 16 Odd 18 Odd

You create an iteration that includes a conditional in the loop body. You can tell the computer to print all the even values between 0 and 19 and state “Odd” in the places where we have odd numbers.

Let’s translate this into computational steps.

If  $x$  leaves a remainder of 0 when divided by 2, which is the same as to say “if  $x$  is even”, then print  $x$  on the same line. “Else”, which means unless  $x$  is even, or if  $x$  is odd, print “Odd”.

## All In – Conditional Statements, Functions, and Loops

We use iterations when we have to go through variables that are part of a list.

You can count the number of items whose value is less than 20 in a list. First, define a function that takes as an argument numbers, where “numbers” will be a certain list variable. The trick is to create a variable that, so to speak, “departs” from 0. Let’s call it total.

```
In [1]: def count(numbers):  
        total = 0  
        for x in numbers:  
            if x < 20:  
                total += 1  
        return total
```

The idea is that, when certain conditions are verified, total will change its value. This is why, in such a situation, it is appropriate to call this variable a **rolling sum**.

More technically, when we consider  $x$  in the numbers list, if it is smaller than 20, we will increment the total by 1 and finally return the total value. This means that, if  $x$  is less than 20, total will grow by 1, and if  $x$  is greater than or equal to 20, total will not grow. So, for a given list, this count function will return the amount of numbers smaller than 20.