

Python Basics I

Why Python Programming Language?

The python language is one of the most accessible programming languages available because it has simplified syntax and is not complicated, which gives more emphasis on natural language. Due to its ease of learning and usage, python codes can be easily written and executed much faster than other programming languages.

Python was created more than 30 years ago, which is a lot of time for any community of programming language to grow and mature adequately to support developers ranging from beginner to expert levels. There are plenty of documentation, guides, and Video Tutorials for Python language that learners and developers of any skill level or ages can use and receive the support required to enhance their knowledge in python programming language.

Due to its corporate sponsorship and big supportive community of python, python has excellent libraries that you can use to select and save your time and effort on the initial cycle of development. There are also lots of cloud media services that offer cross-platform support through library-like tools, which can be extremely beneficial.

Libraries with specific focus are also available like nltk for natural language processing or scikit-learn for machine learning applications.

Variables

Variables are containers for storing data values. Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

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

We declare a name on the left side of the equals operator ("="), and on the right side, we assign the value that we want to save to use later, eg;

```
School = "Hamoye one school"
```

When you create a variable, the line where you assign the value is a step called declaration. We've just declared a variable with a name of "school" and assigned it the value of the string data type "Hamoye one school". This string is now stored in memory, and we're able to access it by calling the variable name "school".

Variable Naming Convention

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

1. A variable name must start with a letter or the underscore character
2. A variable name cannot start with a number
3. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
4. A variable name must not be any reserved word or keyword, e.g. int, name.

Keywords in Python Programming Language

All the keywords except **True**, **False** and **None** are in lowercase and they must be written as they are. The list of all the keywords is given in the image below:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Fig 2.0 Python keyword

5. Variable names are case-sensitive (age, Age and AGE are three different variables)

Examples

First_name = "Zalihat" # string variable

Age = 23 # integer variable

Cgpa = 4.56 # float variable

Present = True # boolean variable

Data Types in python

Almost all languages use data types, they are essential to every program. Data types are how we define values, like words or numbers. If I were to ask you what a sentence is made up of, you would probably reply with "words or characters." Well, in programming, we call them strings. Just the same as we refer to numbers as their own data type as well.

Data types define what we can do and how these values are stored in memory on the computer. We will at some common data types;

1. Integers

These data types are often called integers or *ints*. They are positive or negative WHOLE numbers with no decimal point. Integers are used for a variety of reasons, between math calculations and indexing, eg 30, 2, etc.

2. Floats

Anytime a number has a decimal point on it, they're known as floating point data types. It doesn't matter if it has 1 digit, or 20, it's still a float. The primary use of floats is in math calculations, although they have other uses as well eg 3.5, 4.0, etc.

3. Booleans

The boolean data type is either a True or False value. Think of it like a switch, where it's either off or on. It can't be assigned any other value except for **True** or **False**. Booleans are a key data type, as they provide several uses. One of the most common is for tracking whether something occurred. For instance, if you took a video game and wanted to know if a player was alive, when the player spawned initially, you would set a boolean to "True". When the player lost all their lives, you would set the boolean to "False". This way you can simply check the boolean to see if the player is alive or not. This makes for a quicker program rather than calculating lives each time.

4. Strings

Also known as "String Literals," these data types are the most complex of the four. The actual definition of a string is:

Strings in Python are arrays of bytes representing Unicode characters.

To most beginners, that's just going to sound like a bunch of nonsense, so let's break it down into something simple that we can understand. Strings are nothing more than a set of characters, symbols, numbers, whitespace, and even empty space between two sets of quotation marks. In Python we can use either single or double quotes to create a string. Most of the time it's personal preference, unless you want to include quotes within a string. Whatever is wrapped inside of the quotation marks will be considered a string, even if it's a number, e.g "Hello World!", "3432", etc.

Comments

Comments are like notes that you leave behind, either for yourself or someone else to read. They are not read in by the interpreter, meaning that you can write whatever you want, and the computer will ignore it. A good comment will be short, easy to read, and to the point. Putting a comment on every line is tedious, but not putting any comments at all is bad practice.

Comment starts with a '#' and python ignores anything after the #. Examples of comments

```
print(90) #prints two
```

Python Operators

Operators are used to perform operations on variables and values. Python divides the operators in the following groups:

1. Arithmetic operators
2. Comparison operators
3. Logical operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Name	Operator	Example
Addition	+	$Z = x + y$
Subtraction	-	$Z = x - y$
Multiplication	*	$Z = x * y$
Division	/	$Z = x / y$
Modulus	%	$Z = x \% y$
exponential	**	$Z = x ** 2$

Fig 2.1 Python arithmetic operators

Python Comparison Operators

Comparison operators are used to compare two values:

Name	Operator	Example
Equal	==	$x == y$
Not equal	!=	$y != a$
Greater than	>	$x < y$
Less than	<	$x < y$
Greater than or equal to	>=	$x >= y$
Less than or equal to	<=	$x <= y$

Fig 2.2 Python comparison operators

Python Logical Operators

Logical operators are used to combine conditional statements:

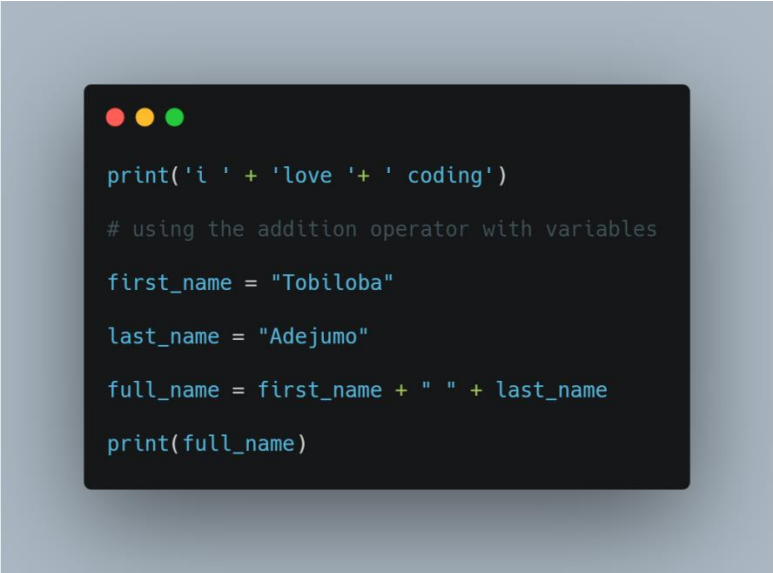
Operator	Description	example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Fig 2.3 Python logical operators

Working with strings

String Concatenation

When we talk about concatenating strings, I mean that we want to add one string to the end of another. This concept is just one of many ways to add string variables together to complete a larger string. For the first example, let's add three separate strings together:



```
print('i ' + 'love ' + ' coding')  
  
# using the addition operator with variables  
  
first_name = "Tobiloba"  
last_name = "Adejumo"  
full_name = first_name + " " + last_name  
print(full_name)
```

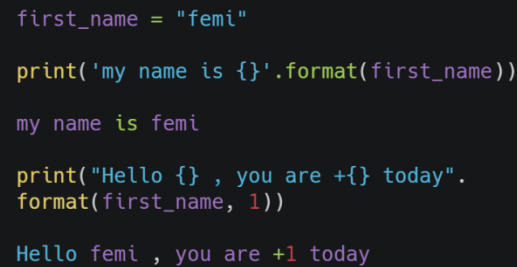
Note: you cannot concatenate a string and a number using the addition operator.

Formatting Strings

Earlier we created a full name by adding multiple strings together to create a larger string. While this is perfectly fine to use, for larger strings it becomes tough to read. Imagine that you had to create a sentence that used 10 variables. Appending all ten variables into a sentence is tough to keep track of, not to mention read. We'll need to use a concept called string formatting. This will allow us to write an entire string and inject the variables we want to use in the proper locations. Using this we can also concatenate a string and a number.

.format()

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays Python code and its output.

```
first_name = "femi"
print('my name is {}'.format(first_name))
my name is femi
print("Hello {} , you are +{} today".format(first_name, 1))
Hello femi , you are +1 today
```

fStrings (New in Python 3.6)

The new way to inject variables into a string in Python is by using what we call f strings. By putting the letter “f” in front of a string, you’re able to inject a variable into a string directly in line. This is important, as it makes the string easier to read when it gets longer, making this the preferred method to format a string. Just keep in mind you need Python 3.6 to use this; otherwise you’ll receive an error. To inject a variable in a string, simply wrap curly brackets around the name of the variable. Let’s look at an example:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays Python code using f-strings and its output.

```
first_name = "Godwin"
print(f"hello {first_name}")
hello Godwin
```

String Index

One other key concept that we need to understand about strings is how they are stored. When a computer saves a string into memory, each character within the string is assigned what we call an “index.” An index is essentially a location in memory. Think of an index as a position in a line that you’re waiting in at the mall. If you were at the front of the line, you would be given an index number of zero. The person behind you would be given index position one. The person behind them would be given index position two and so on.

Note that in python and most programming languages index starts from 0 not 1.

Here are examples of string indexing:

```
word = "animal"
print(word[0])
print(word[1])
print(word[2])
```

As you can see from the example above, you index a string using a square bracket and the index.

Be very careful when working with indexes. An index is a specific location in memory. If you try to access a location that is out of range, you will crash your program because it’s trying to access a place in memory that does not exist.

String Slicing

I want to just quickly introduce the topic of slicing. Slicing is used mostly with Python lists; however, you can use it on strings as well. Slicing is essentially when you only want a piece of the variable, such that if I only wanted “He” from the word “Hello”, we would write the following:

```
print(word[0:2])
```

The first number in the bracket is the starting index; the second is the stopping index. We will touch on this concept later when we talk about lists.


String Manipulation

In many programs that you’ll build, you’re going to want to alter strings in one way or another. String manipulation just means that we want to alter what the current string is.

Luckily, Python has plenty of methods that we can use to alter string data types.

1. **.title()**

Often, you’ll run into words that aren’t capitalized that should be usually names. The title method capitalizes all first letters in each word of a string.



```
first_name = "ayomide"  
print(first_name.title())
```

2. `.upper()`

This method converts a string to uppercase.



```
print("LOve aNd LiGhT".upper())
```

3. `.lower()`


This is the opposite of the `.upper()`, it converts a given string to lowercase.



```
print("LOve aNd LiGhT".lower())
```


4. `.replace()`

The `replace` method works like a find and replace tool. It takes in two values within its parenthesis, one that it searches for and the other that it replaces the searched value with:

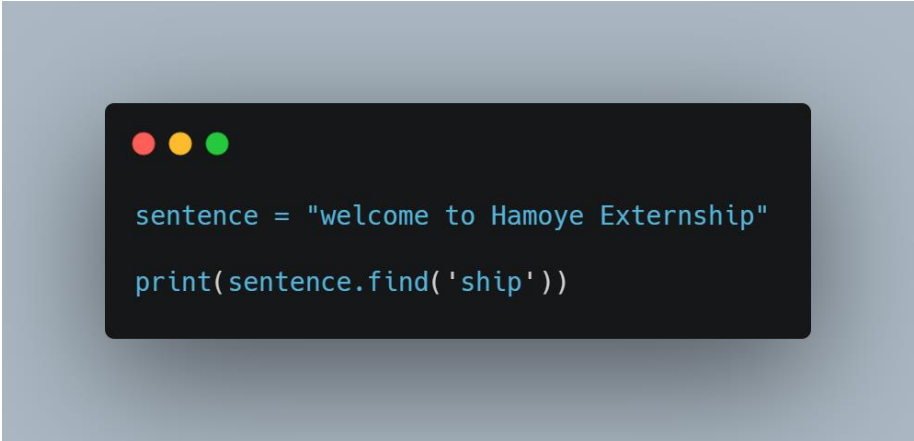


```
word = "Hello there!"  
print(word.replace('!', '.'))
```

5. `.find()`

The `find` method will search for any string we ask it to. In this example, we try to search for an entire word, but we could search for anything including a character or a full

Sentence:



```
sentence = "welcome to Hamoye Externship"  
print(sentence.find('ship'))
```

Note that the method returns the index of the word in the sentence.

In this course we only cover a few string methods. As we move forward we are going to use more methods, to learn more about string methods check the documentation here [3.6.1 String Methods](#).

Conditional Statements

In Python, the code executes in a sequential manner i.e. the first line will be executed first followed by the second line and so on. What will we do in a case where we have to decide that a certain part of code should run only if the condition is True? In such cases, Decision making is required, which is achieved through conditional statements.

If Statement

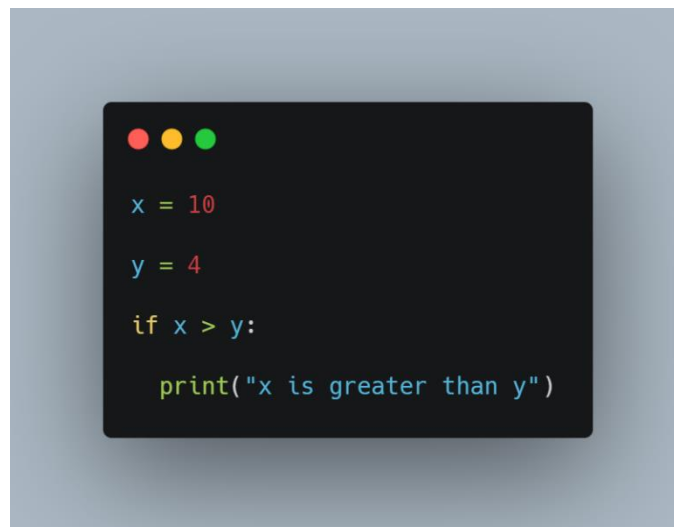
In Python, if statement is used when we have to execute a code block, only if a test condition is True. In this case, the program will evaluate the test expression and will execute statement(s) only if the text expression is True. An if statement is written by using the if keyword.

Syntax:

If (condition):

Statement

Example:



If else Statement

In Python, when we combine the else code block with the if code block, the if code block is executed only if the test condition is True, and the else code block is executed in the cases when the test condition is False. An if else statement is written by using the if else keyword.

Syntax:

If (condition):

statement(s)

Else:

statement(s)

Example:

```
x = 1
y = 4
if x > y:
    print("x is greater than y")
else:
    print("y is greater than x")
```

Elif Statement

In Python, an elif statement is used when we have to check multiple conditions. elif is short form for else if. In such cases, firstly, the if test condition is checked, If it is true, then the if code block is executed and if it is false, then the next elif test condition is checked and so on. If all the conditions are false, then the else code block is executed. An if else statement is written by using the if elif else keyword.

Syntax:

if (condition):

statement(s)

elif (condition):

statement(s)

else:

statement(s)

Example:

```
x = 1
y = 3
if x > y:
    print("x is greater than y")
elif x==y:
    print("x is equal to y")
else:
    print("y is greater than x")
```

Lists

A list is a data structure in Python that is a mutable, ordered sequence of elements.

Mutable means that you can change the items inside, while ordered sequence is in reference to index location. The first element in a list will always be located at index 0.

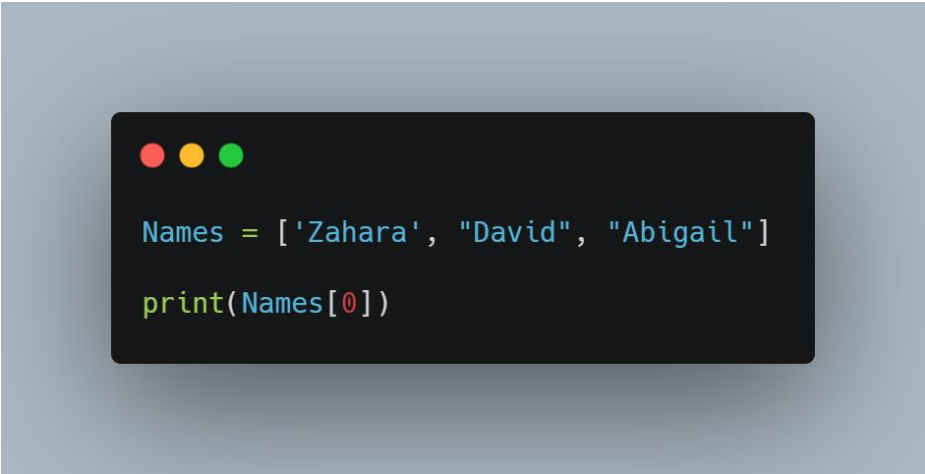
Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having different data types between square brackets []. Also, like strings, each item within a list is assigned an index, or location, for where that item is saved in memory. Lists are also known as a data collection. Data collections are simply data types that can store multiple items. We'll see other data collections, like dictionaries and tuples, later.

Creating a list is just like creating a variable, here is an example,

```
Names = ['Zahara', "David", "Abigail"]
```

Accessing an item within a list

Just like we did with strings, we use a square bracket and an index. Here is an example,

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains two lines of Python code:

```
Names = ['Zahara', "David", "Abigail"]
```

 and

```
print(Names[0])
```

```
Names = ['Zahara', "David", "Abigail"]  
print(Names[0])
```

The above code returns the first item in the list.

You can also create a list with items of different data types. Let's see an example,

```
list1 = [2, 9.7, "word", True]
```

Lists within lists

Let's get a little more complex and see how lists can be stored within another list:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains two lines of Python code:

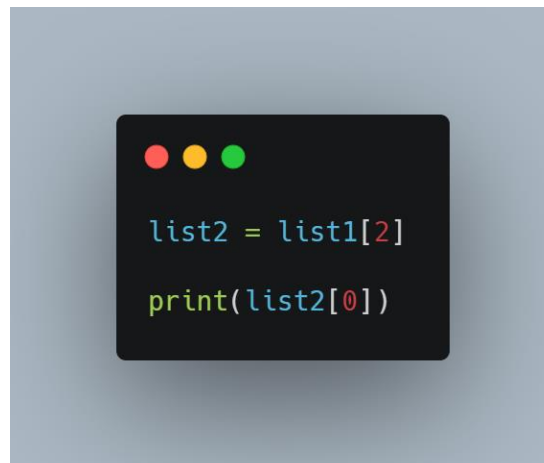
```
list2 = ["Cat", "Dog", ["Fish", "crocodile"], 70]
```

 and

```
print(list2[2])
```

```
list2 = ["Cat", "Dog", ["Fish", "crocodile"], 70]  
print(list2[2])
```

Note that you can access the element in the inner list using index, lets print “fish”,

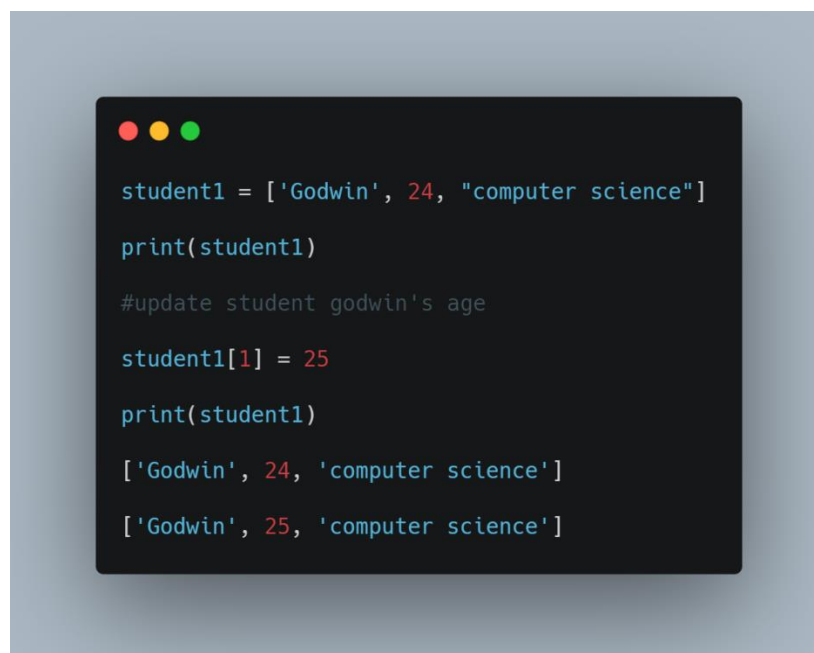


```
list2 = list1[2]
print(list2[0])
```

Altering/ changing the values of items in a list.

At some point we might need to change or update a list. For example' if we have a student record we might need to change the ages every year.

Here is an example;



```
student1 = ['Godwin', 24, "computer science"]
print(student1)

#update student godwin's age
student1[1] = 25
print(student1)

['Godwin', 24, 'computer science']
['Godwin', 25, 'computer science']
```

Python Basics II

List functions

Python offers the following list functions:


1. `sort()`: Sorts the list in ascending order.
2. `type(list)`: It returns the class type of an object.
3. `append()`: Adds a single element to a list.
4. `extend()`: Adds multiple elements to a list.
5. `index()`: Returns the first appearance of the specified value.
6. `max(list)`: It returns an item from the list with max value.
7. `min(list)`: It returns an item from the list with min value.
8. `len(list)`: It gives the total length of the list.
9. `list(seq)`: Converts a tuple into a list.
10. `cmp(list1, list2)`: It compares elements of both lists list1 and list2.

sort() method

The `sort()` method is a built-in Python method that, by default, sorts the list in ascending order. However, you can modify the order from ascending to descending by specifying the sorting criteria.

Example

Let's say you want to sort the element in prices in ascending order. You would type prices followed by a `.` (period) followed by the method name, i.e., `sort` including the parentheses.



```
prices = [23.5, 4, 2.5, 9.0 ]  
prices.sort()  
print(prices)
```

type() method

For the `type()` function, it returns the class type of an object.



```
list1 = [2, 9.7, "word", True]
print(type(list1))
```

append() method

The `append()` method will add certain content you enter to the end of the elements you select.



```
weeks = ["Monday", "Tuesday", "Wednesday"]
weeks.append("Thursday")
print(weeks)
```

extend() method


The `extend()` method increases the length of the list by the number of elements that are provided to the method, so if you want to add multiple elements to the list, you can use this method.



```
weeks = ["Monday", "Tuesday", "Wednesday"]
weeks.extend(["Thursday", "Friday"])
print(weeks)
```

index() method

The index() method returns the first appearance of the specified value.

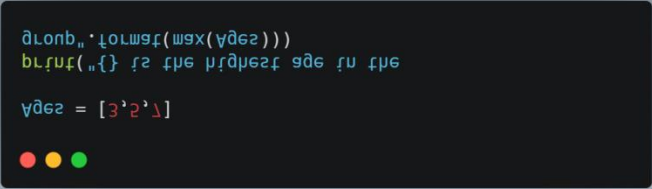
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Python code:

```
weeks = ["Monday", "Tuesday", "Wednesday"]  
print(weeks.index("Monday"))
```

```
weeks = ["Monday", "Tuesday", "Wednesday"]  
print(weeks.index("Monday"))
```

max() function

The max() function will return the highest value of the inputted values.

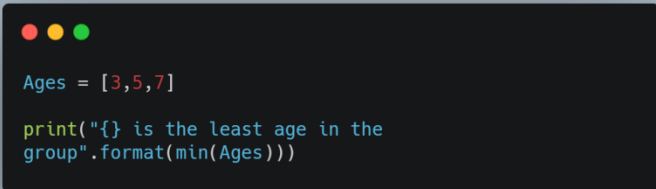
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Python code:

```
monday = format(max(ages))  
print("{} is the highest age in the  
ages = [3,2,1]
```

```
monday = format(max(ages))  
print("{} is the highest age in the  
ages = [3,2,1]
```

min() function

The min() function will return the lowest value of the inputted values.

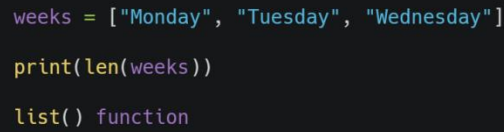
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Python code:

```
Ages = [3,5,7]  
print("{} is the least age in the  
group".format(min(Ages)))
```

```
Ages = [3,5,7]  
print("{} is the least age in the  
group".format(min(Ages)))
```

len() function

The len() function shows the number of elements in a list.



```
weeks = ["Monday", "Tuesday", "Wednesday"]  
print(len(weeks))  
list() function
```

The `list()` function takes an iterable construct and turns it into a list.



```
string1 = "python"  
list1 = list(string1)  
print(list1)  
print(list1[4])
```

To learn more about list methods and functions, check the documentation below

[5. Data Structures — Python 3.9.0 documentation](#)

Slicing list

Tuples

Tuples are an ordered sequence of items, just like lists. The main difference between tuples and lists is that tuples cannot be changed (immutable) unlike lists which can be (mutable). When you try to change a value in a tuple it will throw a type error. Similarly, you cannot add or remove values either.

There are two ways to declare or initialize a tuple.

To create a tuple you use parentheses instead of square brackets, and just provide it the values separated by commas.

1. You can initialize an empty tuple by having `()` with no values in them.

```
empty_tuple = ()
```

2. You can also initialize an empty tuple by using the tuple function.

```
empty_tuple = tuple()
```

Accessing elements of a tuple is just like accessing lists, you use indexes. Let's take a look at an example;

```
scores = (2.4, 6.9, 3.6, 8.3, 2.0, 4.5)
```

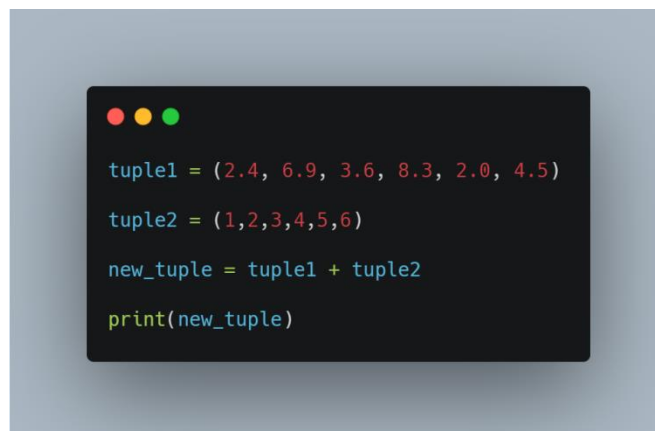
```
print(scores[4])
```

Note that you cannot add items to a tuple, it will throw an error. Let's try this



The screenshot shows a Jupyter Notebook interface. The first cell contains the code `scores = (2.4, 6.9, 3.6, 8.3, 2.0, 4.5)` and `print(scores[4])`, which has been executed, resulting in the output `2.0`. The second cell contains the code `scores[6] = 3`, which has been executed and resulted in a `TypeError`. The error message is: `TypeError: 'tuple' object does not support item assignment`. The traceback shows the error occurred in the current cell at line 1.

You can add two tuples together to create a new tuple,



```
tuple1 = (2.4, 6.9, 3.6, 8.3, 2.0, 4.5)
tuple2 = (1, 2, 3, 4, 5, 6)
new_tuple = tuple1 + tuple2
print(new_tuple)
```

Note:

You can't add elements to a tuple because of their immutable property.

There's no `append()` or `extend()` method for tuples, You can't remove elements from a tuple.

Also because of their immutability, Tuples have no `remove()` or `pop()` method,

You can find elements in a tuple, since this doesn't change the tuple.

You can also use the `in` operator to check if an element exists in the tuple.

You can delete a tuple by using the `del` keyword followed by the name of the tuple.

```
del tuple1
```

Sets

Python set is an unordered collection of unique items. They are commonly used for computing mathematical operations such as union, intersection, difference, and symmetric difference.

The important properties of Python sets are as follows:

1. Sets are unordered – Items stored in a set aren't kept in any particular order.
2. Set items are unique – Duplicate items are not allowed.
3. Sets are unindexed – You cannot access set items by referring to an index.
4. Sets are changeable (mutable) – They can be changed in place, can grow and shrink on demand.

You can create a set by placing a comma-separated sequence of items in curly brackets `{}`.

Note that if you create a set with duplicate items, the duplicate will be removed automatically during the set creation.

A screenshot of a code editor window with a dark background and light-colored text. The code defines a set named 'colors' with the elements 'red', 'black', 'white', 'green', and 'black'. The duplicate 'black' is automatically removed. The code then prints the set.

```
colors = {"red", "black", "white", "green", "black"}  
print(colors)
```

You can also create a set using the `set` function, this is also great for converting sequence/ iterable to set. For example; let's convert a list to a set.

```
● ● ●  
  
colors = set(["red", "black", "white", "green",  
"black"])  
  
print(colors)
```

This is also a great way to remove duplicates from an iterable, you can simply convert the iterable to a set which removes the duplicates, and then you convert it back to the iterable type.

```
● ● ●  
  
colors = set(["red", "black", "white", "green",  
"black"])  
  
colors = list(colors)  
  
print(colors)
```

Adding items to a set

```
● ● ●  
  
# using the add method  
  
colors = set(["red", "black", "white", "green",  
"black"])  
  
colors.add("Blue")  
  
print(colors)
```

To add an element to a set, you use the add method; and to **add** multiple elements you use the **update** method.

Remove Items from a Set

To remove a single item from a set, use **remove()** or **discard()** method.

```
# using the remove method

colors = set(["red", "black", "white", "green",
"black"])

colors.remove('red')

print(colors)
```

Find Set Size

To find how many items a set has, use **len()** method.

```
colors = set(["red", "black", "white", "green"])

print(len(colors))
```

Set Operations

Sets are commonly used for computing mathematical operations such as intersection, union, difference, and symmetric difference.

Set Union

You can perform union on two or more sets using **union()** method or **|** operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator
print(A | B)

# Prints {'blue', 'green', 'yellow', 'orange', 'red'}

# by method
print(A.union(B))

# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

Set Intersection

You can perform intersection on two or more sets using **intersection()** method or **&** operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator
print(A & B)

# Prints {'red'}

# by method
print(A.intersection(B))

# Prints {'red'}
```

Set Difference

You can compute the difference between two or more sets using **difference()** method or **-** operator.

```
● ● ●

A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator

print(A - B)

# Prints {'blue', 'green'}

# by method

print(A.difference(B))

# Prints {'blue', 'green'}
```

You can compute symmetric difference between two or more sets using **symmetric_difference()** method or **^** operator.

```
● ● ●

A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator

print(A ^ B)

# Prints {'orange', 'blue', 'green', 'yellow'}

# by method

print(A.symmetric_difference(B))

# Prints {'orange', 'blue', 'green', 'yellow'}
```

Other Set Operations

Below is a list of all set operations available in Python.

Method & Their Description

union() Return a new set containing the union of two or more sets

update() Modify this set with the union of this set and other sets

intersection() Returns a new set which is the intersection of two or more sets

intersection_update() Removes the items from this set that are not present in other sets

difference() Returns a new set containing the difference between two or more sets

difference_update() Removes the items from this set that are also included in another set

symmetric_difference() Returns a new set with the symmetric differences of two or more sets

symmetric_difference_update() Modify this set with the symmetric difference of this set and other set

isdisjoint() Determines whether or not two sets have any elements in common

issubset() Determines whether one set is a subset of the other

issuperset() Determines whether one set is a superset of the other

Python Basics III

Dictionaries

Dictionaries are Python's implementation of a data structure, generally known as associative arrays, hashes, or hashmaps.

You can think of a dictionary as a mapping between a set of indexes (known as keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key:value pair or sometimes an item.

You can create a dictionary by placing a comma-separated list of key:value pairs in curly braces {}. Each key is separated from its associated value by a colon :



```
employee1 = {'name': 'Manasseh',  
             'age': 23,  
             'job': 'DevOps',  
             'city': 'Jos',  
             'email': 'manasseh@web.com'}  
  
print(employee1)
```

There are lots of other ways to create a dictionary.

You can use dict() function along with the zip() function, to combine separate lists of keys and values obtained dynamically at runtime.



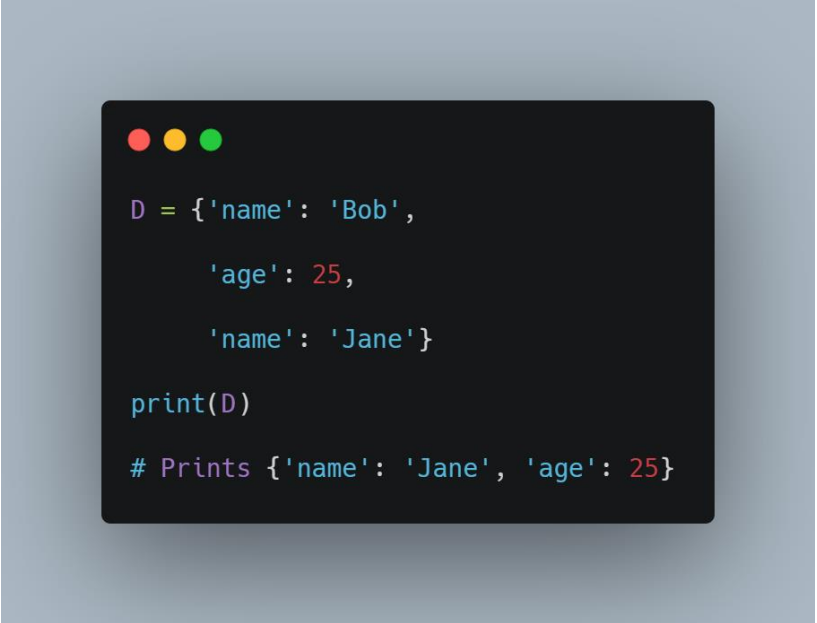
```
# Create a dictionary with list of zipped keys/values  
keys = ['name', 'age', 'job']  
values = ['Bob', 25, 'Dev']  
D = dict(zip(keys, values))  
print(D)  
  
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

Important Properties of a Dictionary

Dictionaries are pretty straightforward, but here are a few points you should be aware of when using them.

1. Keys must be unique:
2. A key can appear in a dictionary only once.

Even if you specify a key more than once during the creation of a dictionary, the last value for that key becomes the associated value.



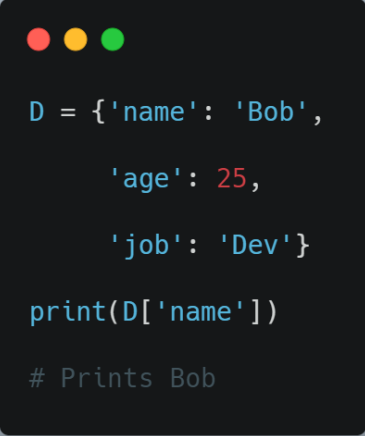
```
D = {'name': 'Bob',  
     'age': 25,  
     'name': 'Jane'}  
  
print(D)  
  
# Prints {'name': 'Jane', 'age': 25}
```

Access Dictionary Items

The order of key:value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.

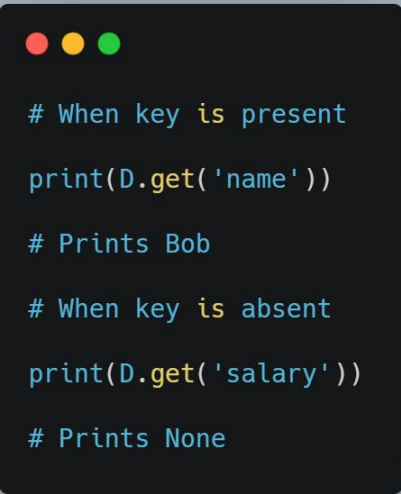
However, this is not a problem because the items of a dictionary are not indexed with integer indices. Instead, you use the keys to access the corresponding values.

You can fetch a value from a dictionary by referring to its key in square brackets [].



```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
print(D['name'])  
  
# Prints Bob
```

If you refer to a key that is not in the dictionary, you'll get an exception. To avoid such exceptions, you can use the special dictionary `get()` method. This method returns the value for key if key is in the dictionary, else `None`, so that this method never raises a `KeyError`.



```
# When key is present  
print(D.get('name'))  
  
# Prints Bob  
  
# When key is absent  
print(D.get('salary'))  
  
# Prints None
```

Add or Update Dictionary Items.

Adding or updating dictionary items is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, its value is replaced by the new one.

```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
D['name'] = 'Sam'  
  
print(D)  
  
# Prints {'name': 'Sam', 'age': 25, 'job': 'Dev'}
```

If the key is new, it is added to the dictionary with its value.

```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
D['city'] = 'New York'  
  
print(D)  
  
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev',  
          'city': 'New York'}
```

Merge Two Dictionaries

Use the built-in `update()` method to merge the keys and values of one dictionary into another. Note that this method blindly overwrites values of the same key if there's a clash.

```
D1 = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
D2 = {'age': 30,  
      'city': 'New York',  
      'email': 'bob@web.com'}  
  
D1.update(D2)  
  
print(D1)  
  
# Prints {'name': 'Bob', 'age': 30, 'job': 'Dev',  
#        'city': 'New York', 'email': 'bob@web.com'}
```

Remove Dictionary Items

There are several ways to remove items from a dictionary.

1. Remove an Item by Key

If you know the key of the item you want, you can use the `pop()` method. It removes the key and returns its value.

```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
x = D.pop('age')  
  
print(D)  
  
# Prints {'name': 'Bob', 'job': 'Dev'}  
  
# get removed value  
  
print(x)  
  
# Prints 25
```

2. If you don't need the removed value, use the del statement.

```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
del D['age']  
  
print(D)  
  
# Prints {'name': 'Bob', 'job': 'Dev'}
```

3. Remove all Items

To delete all keys and values from a dictionary, use the clear() method.

```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
D.clear()  
  
print(D)  
  
# Prints {}
```

While Loop

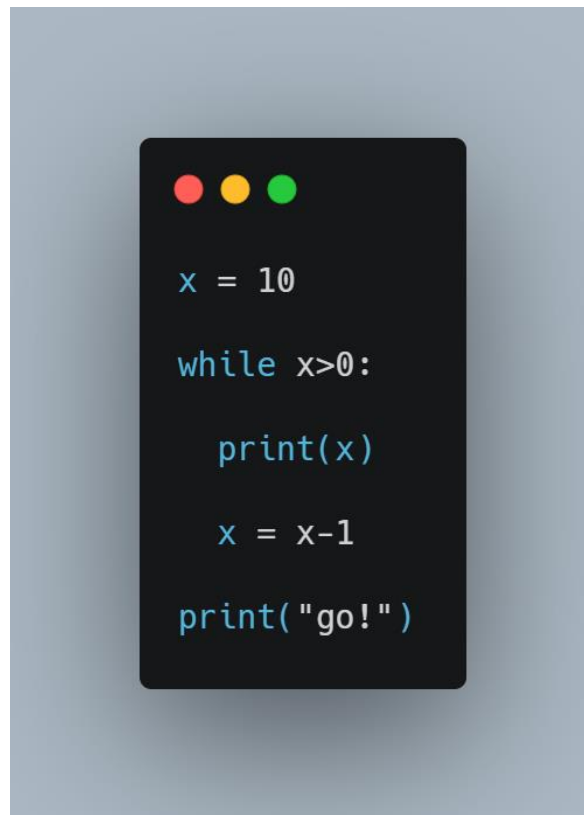
The while loop is somewhat similar to an if statement, it executes the code inside, if the condition is True. However, as opposed to the if statement, the while loop continues to execute the code repeatedly as long as the condition is True. Repeating an action until a condition is met

Syntax:

While condition:

statement(s)

Example, a program for counting down from 10 to 0

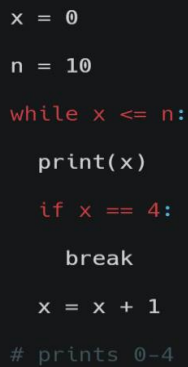


So here is what is happening in this example. The loop starts by checking if the condition is met, since x is greater than 0 it prints x which is 10, after that x is reduced by 1, this will continue until the condition is no longer true, i.e when x is less than 0 and it prints go!

The Python Break and Continue Statements

In the above example, the entire body of the while loop is executed on each iteration. Python provides two keywords that terminate a loop iteration prematurely:

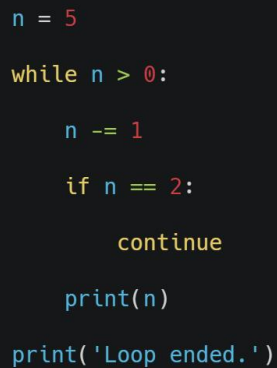
The Python break statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted font. It defines a variable x as 0 and n as 10. A while loop with condition x <= n contains a print statement for x, an if statement that checks if x is 4 and breaks the loop, and an increment of x by 1. A comment at the bottom indicates the output will be 0-4.

```
x = 0
n = 10
while x <= n:
    print(x)
    if x == 4:
        break
    x = x + 1
# prints 0-4
```

In the above example, the program execution is stopped when n is equal to 4.

The Python continue statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted font. It defines a variable n as 5. A while loop with condition n > 0 contains a decrement of n by 1, an if statement that checks if n is 2 and continues the loop, a print statement for n, and a final print statement at the end of the loop that says 'Loop ended.'.

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

This time, when n is 2, the continue statement causes termination of that iteration. Thus, 2 isn't printed. Execution returns to the top of the loop, the condition is re-evaluated, and it is still true. The loop resumes, terminating when n becomes 0, as previously.

Python Functions

Functions

Functions are the first step to code reuse. They allow you to define a reusable block of code that can be used repeatedly in a program.

Python provides several built-in functions such as `print()`, `len()` or `type()`, but you can also define your own functions to use within your programs.

Syntax

The basic syntax for a Python function definition is:

`Def function_name(arguments):`

Statement

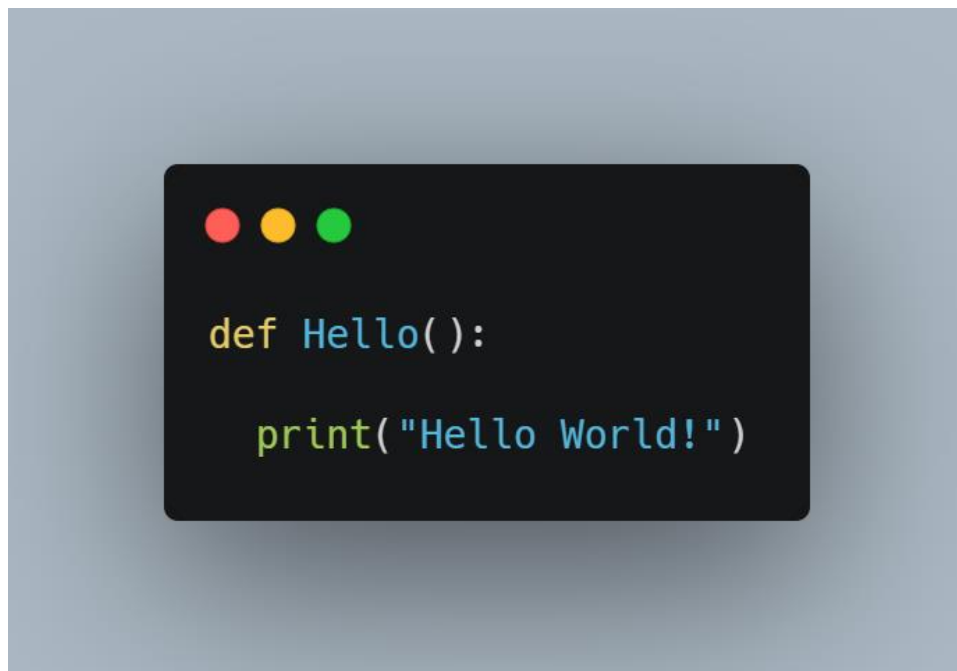
Return value

Function_name is the name of the function you wish to define, it has the same naming convention as a variable.

Statement is the function's body, it is executed whenever the function is called.

Return value ends the function call and sends data back to the program.

To define a Python function, use the `def` keyword. Here's the simplest possible function that prints 'Hello, World!' on the screen.



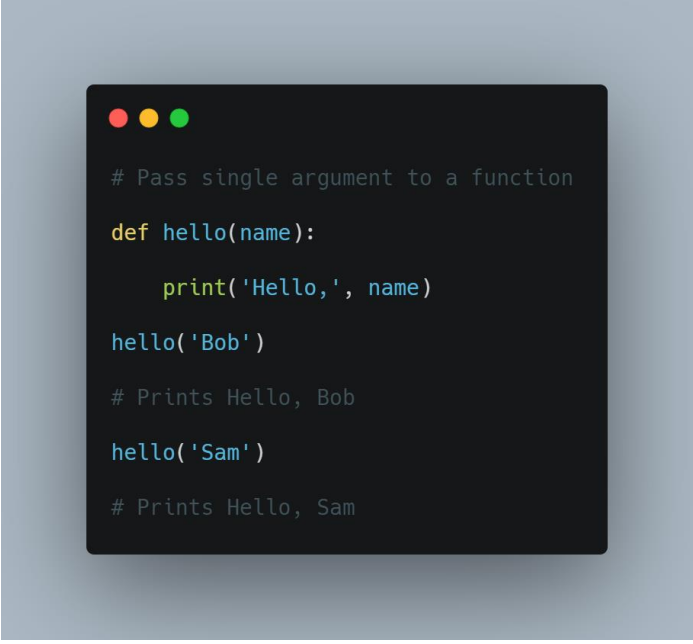
The `def` statement only creates a function but does not call it. After the `def` has run, you can call (run) the function by adding parentheses after the function's name.

Hello()

Pass Arguments

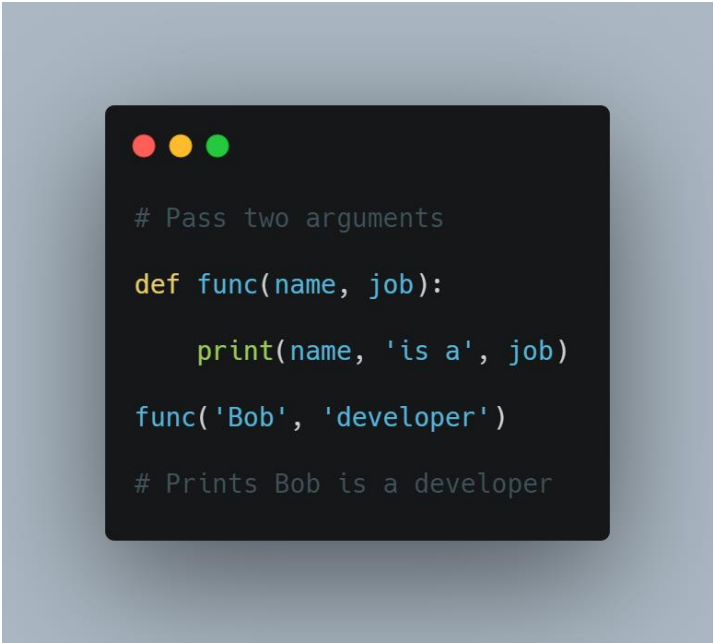
You can send information to a function by passing values, known as arguments. Arguments are declared after the function name in parentheses.

When you call a function with arguments, the values of those arguments are copied to their corresponding parameters inside the function.



```
# Pass single argument to a function
def hello(name):
    print('Hello,', name)
hello('Bob')
# Prints Hello, Bob
hello('Sam')
# Prints Hello, Sam
```

You can send as many arguments as you like, separated by commas ,.



```
# Pass two arguments
def func(name, job):
    print(name, 'is a', job)
func('Bob', 'developer')
# Prints Bob is a developer
```

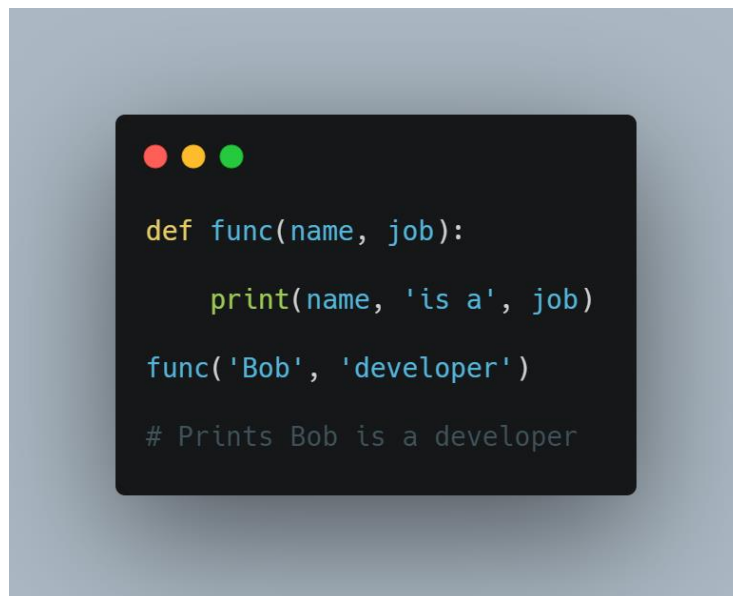
Types of Arguments

Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition. Here's the list:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable Length Positional Arguments (*args)
5. Variable Length Keyword Arguments (**kwargs)

Positional Arguments

The most common are positional arguments, whose values are copied to their corresponding parameters in order.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code inside is as follows:


```
def func(name, job):  
    print(name, 'is a', job)  
  
func('Bob', 'developer')  
  
# Prints Bob is a developer
```

The only downside of positional arguments is that you need to pass arguments in the order in which they are defined.

Keyword Arguments

To avoid positional argument confusion, you can pass arguments using the names of their corresponding parameters.

In this case, the order of the arguments no longer matters because arguments are matched by name, not by position.



```
# Keyword arguments can be put in any order

def func(name, job):

    print(name, 'is a', job)

func(name='Bob', job='developer')

# Prints Bob is a developer

func(job='developer', name='Bob')

# Prints Bob is a developer
```

Default Arguments

You can specify default values for arguments when defining a function. The default value is used if the function is called without a corresponding argument.

In short, defaults allow you to make selected arguments optional.



```
# Set default value 'developer' to a 'job' parameter

def func(name, job='developer'):

    print(name, 'is a', job)

func('Bob', 'manager')

# Prints Bob is a manager

func('Bob')

# Prints Bob is a developer
```

*Variable Length Arguments (*args and **kwargs)*

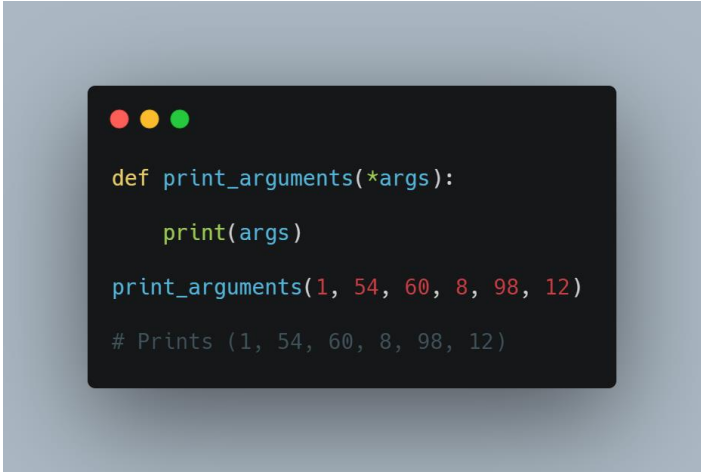
Variable length arguments are useful when you want to create functions that take an unlimited number of arguments. Unlimited in the sense that you do not know beforehand how many arguments can be passed to your function by the user.

This feature is often referred to as var-args.

***args**

When you prefix a parameter with an asterisk `*`, it collects all the unmatched positional arguments into a tuple. Due to the fact that it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.

Following function prints all the arguments passed to the function as a tuple.



```
def print_arguments(*args):  
    print(args)  
  
print_arguments(1, 54, 60, 8, 98, 12)  
  
# Prints (1, 54, 60, 8, 98, 12)
```

You don't need to call this keyword parameter `args`, but it is standard practice.

****kwargs**

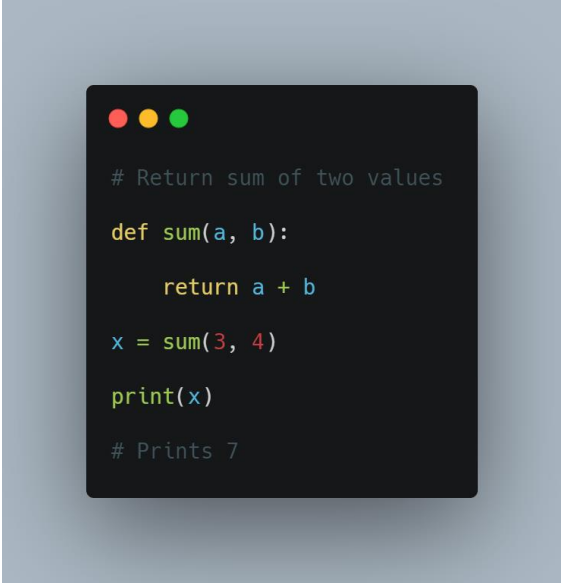
The `**` syntax is similar, but it only works for keyword arguments. It collects them into a new dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.



```
def print_arguments(**kwargs):  
    print(kwargs)  
  
print_arguments(name='Bob', age=25, job='dev')  
  
# Prints {'name': 'Bob', 'age': 25, 'job': 'dev'}
```

Return Value

To return a value from a function, simply use a `return` statement. Once a `return` statement is executed, nothing else in the function body is executed.



```
# Return sum of two values

def sum(a, b):
    return a + b

x = sum(3, 4)


print(x)

# Prints 7
```

Remember! a python function always returns a value. So, if you do not include any return statement, it automatically returns None.

Return Multiple Values

Python has the ability to return multiple values, something missing from many other languages. You can do this by separating return values with a comma.



```
# Return addition and subtraction in a tuple

def func(a, b):
    return a+b, a-b

result = func(3, 2)

print(result)

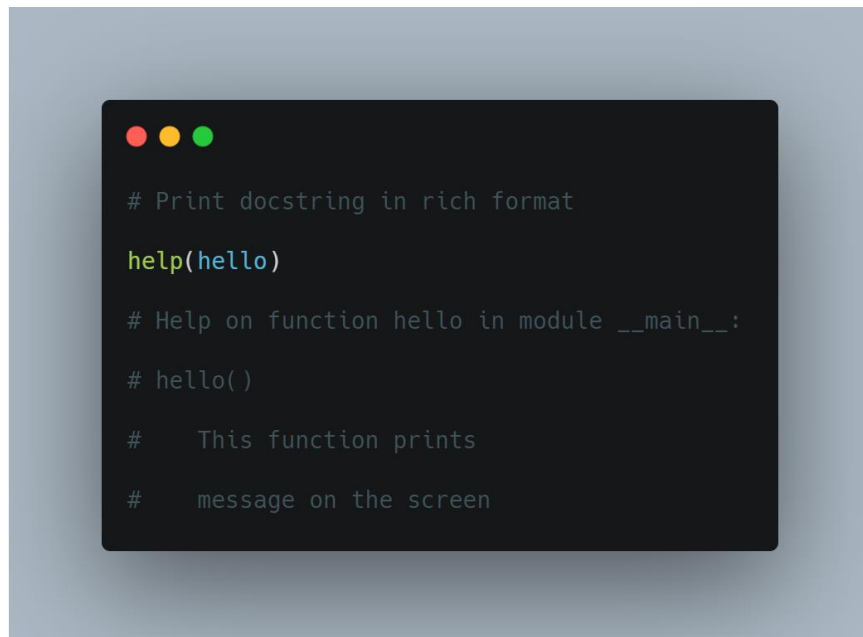
# Prints (5, 1)
```

Docstring

You can attach documentation to a function definition by including a string literal just after the function header. Docstrings are usually triple quoted to allow for multi-line descriptions.

```
def hello():  
    """This function prints  
       message on the screen"""  
    print('Hello, World!')
```

To print a function's docstring, use the Python `help()` function and pass the function's name.



```
# Print docstring in rich format  
help(hello)  
  
# Help on function hello in module __main__:  
  
# hello()  
  
#     This function prints  
#     message on the screen
```

Python Classes

Classes

Classes and objects are the two main aspects of object-oriented programming.

A class is the blueprint from which individual objects are created. In the real world, for example, there may be thousands of cars in existence, all of the same make and model.

Each car was built from the same set of blueprints, and therefore contains the same components. In object-oriented terms, we say that your car is an instance (object) of the class Car.

Creating a Class

To create your own custom object in Python, you first need to define a class, using the keyword `class`.

Suppose you want to create objects to represent information about cars. Each object will represent a single car. You'll first need to define a class called Car.

Here's the simplest possible class (an empty one):

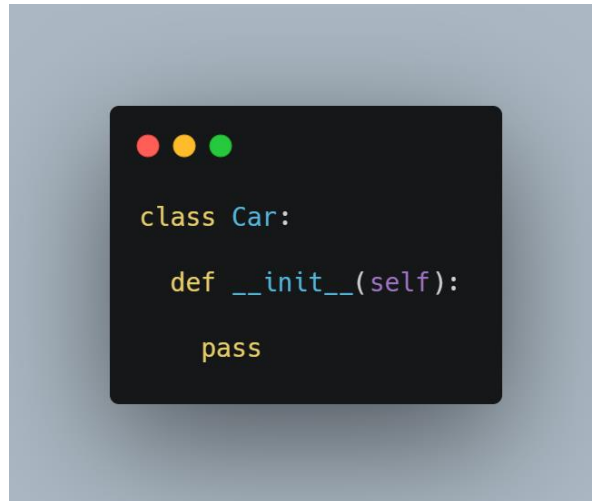


Here the `pass` statement is used to indicate that this class is empty.

The `__init__()` Method

`__init__()` is the special method that initializes an individual object. This method runs automatically each time an object of a class is created.

The `__init__()` method is generally used to perform operations that are necessary before the object is created.



Whenever you define an init method, its first parameter should be self.

The self Parameter

The self parameter refers to the individual object itself. It is used to fetch or set attributes of the particular instance.

This parameter doesn't have to be called self, you can call it whatever you want, but it is standard practice, and you should probably stick with it.

Attributes

Every class you write in Python has two basic features: attributes and methods.

Attributes are the individual things that differentiate one object from another. They determine the appearance, state, or other qualities of that object.

In our case, the 'Car' class might have the following attributes:

Style: Sedan, SUV, Coupe

Color: Silver, Black, White

Wheels: Four

Attributes are defined in classes by variables, and each object can have its own values for those variables.

There are two types of attributes: Instance attributes and Class attributes

1. Instance Attribute

The instance attribute is a variable that is unique to each object (instance). Every object of that class has its own copy of that variable. Any changes made to the variable don't reflect in other objects of that class.

In the case of our Car() class, each car has a specific color and style.

```
# A class with two instance attributes

class Car:

    # initializer with instance attributes

    def __init__(self, color, style):

        self.color = color

        self.style = style
```

2. Class Attribute

The class attribute is a variable that is the same for all objects. And there's only one copy of that variable that is shared with all objects. Any changes made to that variable will reflect in all other objects.

In the case of our Car() class, each car has 4 wheels.

```
# A class with one class attribute

class Car:

    # class attribute

    wheels = 4

    # initializer with instance attributes

    def __init__(self, color, style):

        self.color = color

        self.style = style
```

So while each car has a unique style and color, every car will have 4 wheels.

Creating an Object

You create an object of a class by calling the class name and passing arguments as if it were a function.

Here is an example,

```

# Create an object from the 'Car' class by passing
style and color

class Car:

    # class attribute

    wheels = 4

    # initializer with instance attributes

    def __init__(self, color, style):

        self.color = color

        self.style = style

c = Car('Sedan', 'Black')

```

Here, we created a new object from the Car class by passing strings for the style and color parameters. But, we didn't pass the self argument. This is because, when you create a new object, Python automatically determines what self is (our newly-created object in this case) and passes it to the __init__ method.

Access and Modify Attributes

The attributes of an instance are accessed and assigned to by using dot . notation.

```

# Access and modify attributes of an object

class Car:

    # class attribute

    wheels = 4

    # initializer with instance attributes

    def __init__(self, color, style):

        self.color = color

        self.style = style

c = Car('Black', 'Sedan')

# Access attributes

print(c.style)

# Prints Sedan

print(c.color)

# Prints Black

# Modify attribute

c.style = 'SUV'

print(c.style)

# Prints SUV

```

Methods

Methods determine what type of functionality a class has, how it handles its data, and its overall behavior. Without methods, a class would simply be a structure.

In our case, the 'Car' class might have the following methods:

1. Change color
2. Start engine
3. Stop engine
4. Change gear

Just as there are instance and class attributes, there are also instance and class methods.

Instance methods operate on an instance of a class; whereas class methods operate on the class itself.

1. Instance Methods

Instance methods are nothing but functions defined inside a class that operates on instances of that class.

Now let's add some methods to the class.

1. showDescription() method: to print the current values of all the instance attributes
2. changeColor() method: to change the value of 'color' attribute

```
class Car:
    # class attribute
    wheels = 4
    # initializer / instance attributes
    def __init__(self, color, style):
        self.color = color
        self.style = style
    # method 1
    def showDescription(self):
        print("This car is a", self.color, self.style)
    # method 2
    def changeColor(self, color):
        self.color = color
c = Car('Black', 'Sedan')
# call method 1
c.showDescription()
# Prints This car is a Black Sedan
# call method 2 and set color
c.changeColor('White')
c.showDescription()
# Prints This car is a White Sedan
```

Delete Attributes and Objects

To delete any object attribute, use the del keyword.

```
del c.color
```

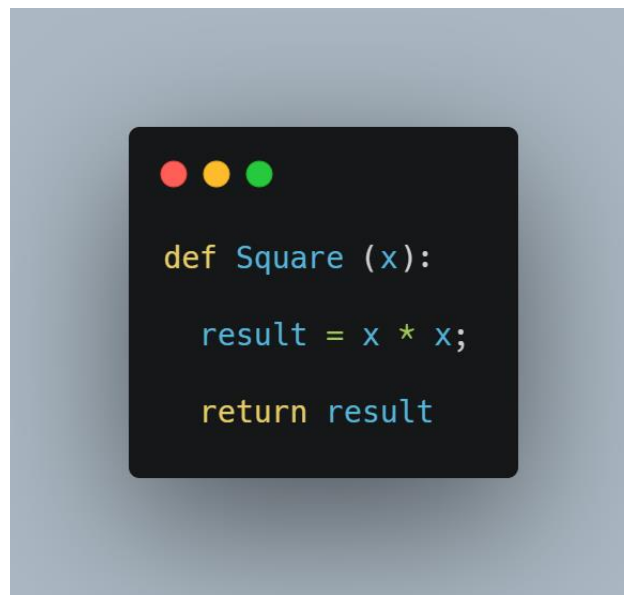
Python Modules

Modules

A module is a piece of software that has a specific functionality. For example, when building a ping pong game, one module would be responsible for the game logic, and another module would be responsible for drawing the game on the screen. Each module is a different file, which can be edited separately.

Modularizing your code is one of the best ways to reuse your code. So let's look at an example, say you want to write a program, and you have to use a square function like more than 100x, remember this is just an example to show you how to create modules in python. I know we have been using Jupyter notebook and colab but as you move forward you will have to use some other editors, but for this simple program we are going to use a text editor.

1. First create a file and name it square_module.py
2. Open jupyter notebook and create a square function, Note that the notebook must be saved in the same folder as the python file. Here is the code to create a square function;



1. Call the function and pass an argument to make sure the function works as expected. For example you can run this code in a new cell,

Square(5)

Note that the expected output is 25.

2. If the code produces the desired result then copy the function code and paste it in the square_module.py file, save and exit.

You have successfully created a python module, i.e the square_module.py file.

Importing a Module

Modules are imported in python using the import statement. In our example above we can import the square.py module by simple running the import square command as shown below;

```
import square_module
```

Here the command imports the entire module, but what we are interested in, is the function in the module. To import a function or class from a module we use the from module import statement, here is how we import the square function;

```
from square_module import square
```

```
square(8)
```

```
#prints 64
```

It is good to know how to create and import modules, but python has some modules that we can use to perform some tasks without having to create them ourselves.

Python has lots of modules that we can use to perform different tasks. Python built in Modules such as the math module has built in functions that can be used to perform math related tasks. For example to find the square root of a number, we don't need to write a function to do that we just need to import the sqrt function in the math module. Here is an example;

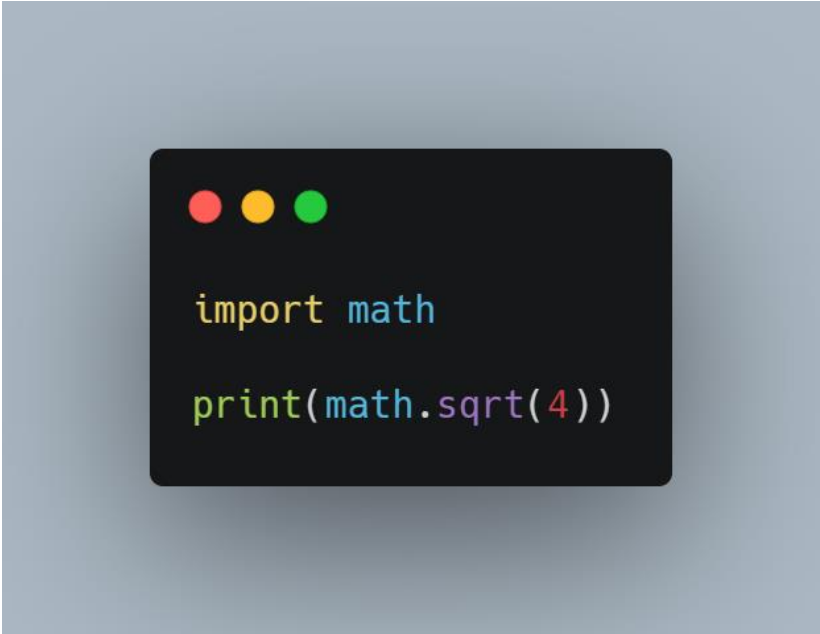
Before the example here are some thing to note, there are two ways to import the function

1. You can import the entire math module and then get the sqrt function from it, or
2. You can import just the sqrt function.

The second option is great for when you need just one function from the module, the first is great for when you need to use more than one function from a module.

Here is how you can do it in code

First option;

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code inside the editor is:

```
import math  
  
print(math.sqrt(4))
```

```
import math  
  
print(math.sqrt(4))
```

we can also use another function like the power function

```
print(math.pow(2,2))
```

Notice the second example, we used the power function from the math module, the math.pow takes two arguments, the number and its power, so for this example we get 4 because 2 to the power of 2 is 4.

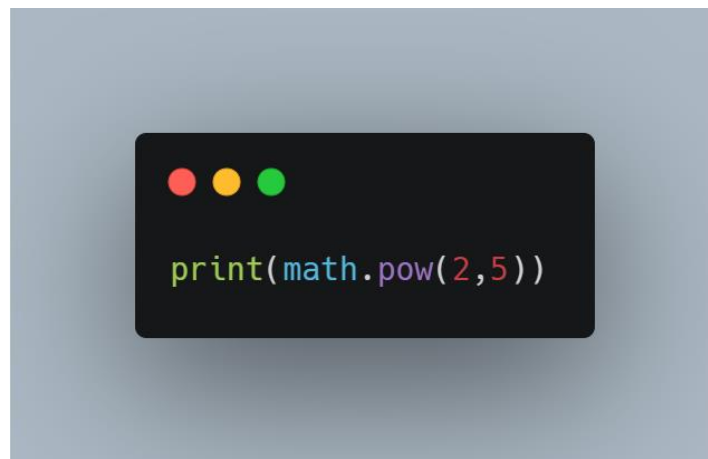
Second option:

```
from math import sqrt
```

```
print(sqrt(4))
```

Restart the runtime , if you are using colab go to runtime and choose restart runtime, and if you are running it locally you go to kernels and choose restart.

Run this in a new cell,



This will give an error because we did not import the entire math module.

There are other modules in python you will get to know them as we move forward.

Python Libraries

Python libraries are different from python modules, a python module is a file which contains python functions , global variables etc. It is nothing but a .py file which has python executable code or statement.

A Python library is a reusable chunk of code that you may want to include in your programs/ projects. Compared to languages like C++ or C, Python libraries do not pertain to any specific context in Python. Here, a 'library' loosely describes a collection of core modules. Essentially, then, a library is a collection of modules. A package is a library that can be installed using a package manager like pip

What is pip?

What is pip? pip is the standard package manager for Python. It allows you to install and manage additional packages that are not part of the Python standard library.

Important libraries in python for data science.

Here are libraries that will take you places in your journey with Python. These are also the Python libraries for Data Science.

1. Pandas

Like we've said before, Pandas is a must for data-science. It provides fast, expressive, and flexible data structures to easily (and intuitively) work with structured (tabular, multidimensional, potentially heterogeneous) and time-series data.

2. NumPy

It has advanced math functions and a rudimentary scientific computing package.

3. Matplotlib

Matplotlib helps with data analyzing, and is a numerical plotting library.

4. SciPy

SciPy has a number of user-friendly and efficient numerical routines. These include routines for optimization and numerical integration.

5. SciKit-Learn

This is an industry-standard for data science projects based in Python. Scikits is a group of packages in the SciPy Stack that were created for specific functionalities – for example, image processing. Scikit-learn uses the math operations of SciPy to expose a concise interface to the most common machine learning algorithms.

Data scientists use it for handling standard machine learning and data mining tasks such as clustering, regression, model selection, dimensionality reduction, and classification. Another advantage? It comes with quality documentation and offers high performance.

6. TensorFlow

TensorFlow is a popular Python framework for machine learning and deep learning, which was developed at Google Brain. It's the best tool for tasks like object identification, speech recognition, and many others. It helps in working with artificial neural networks that need to handle multiple data sets.

7. PyTorch

PyTorch is a framework that is perfect for data scientists who want to perform deep learning tasks easily. The tool allows performing tensor computations with GPU acceleration. It's also used for other tasks – for example, for creating dynamic computational graphs and calculating gradients automatically.

Installing Libraries in python

Now we have seen python libraries and what they can be used for, let's see how we can install and use them. Here we are going to look at two different ways to install libraries in python.

1. Installing libraries in colab

One of the reasons colab is widely used is because it comes with some libraries for data science installed. There are times when you need to install some libraries too.

To import a library that's not in a Colab by default, you can use `!pip install` or `!apt-get install`.

Pandas is installed in colab by default but i will use this to show you how to install libraries.

```
!pip install pandas
```

2. Installing libraries in Anaconda

You can also use the graphical interface Anaconda Navigator to install conda packages with just a few clicks.

Open an Anaconda Prompt (terminal on Linux or macOS) and follow these instructions.

Enter the command:

```
conda install package-name
```

To install pandas you can enter the command;

```
Conda install pandas
```

Note that you can also install packages/ libraries on a jupyter notebook using the pip too,

It uses the same syntax as colab

```
!pip install pandas
```

Importing libraries

To import a library you use the import statement. For example to import the pandas library you run the command;

```
import pandas
```

In python you can import a library and rename it. What this means is that, when you have a library and its name is complex or too long for you to remember you can rename it to something simple or short.

Syntax:

```
Import package_name as new_name
```

Here is an example,

```
import pandas as pd
```

You will notice as we go forward that this is the way most libraries will be imported.