PYTHON

Python Introduction

What is Python?

Python is an interpreted, object-oriented, high-level popular programming language with dynamic semantics . It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Python Quickstart

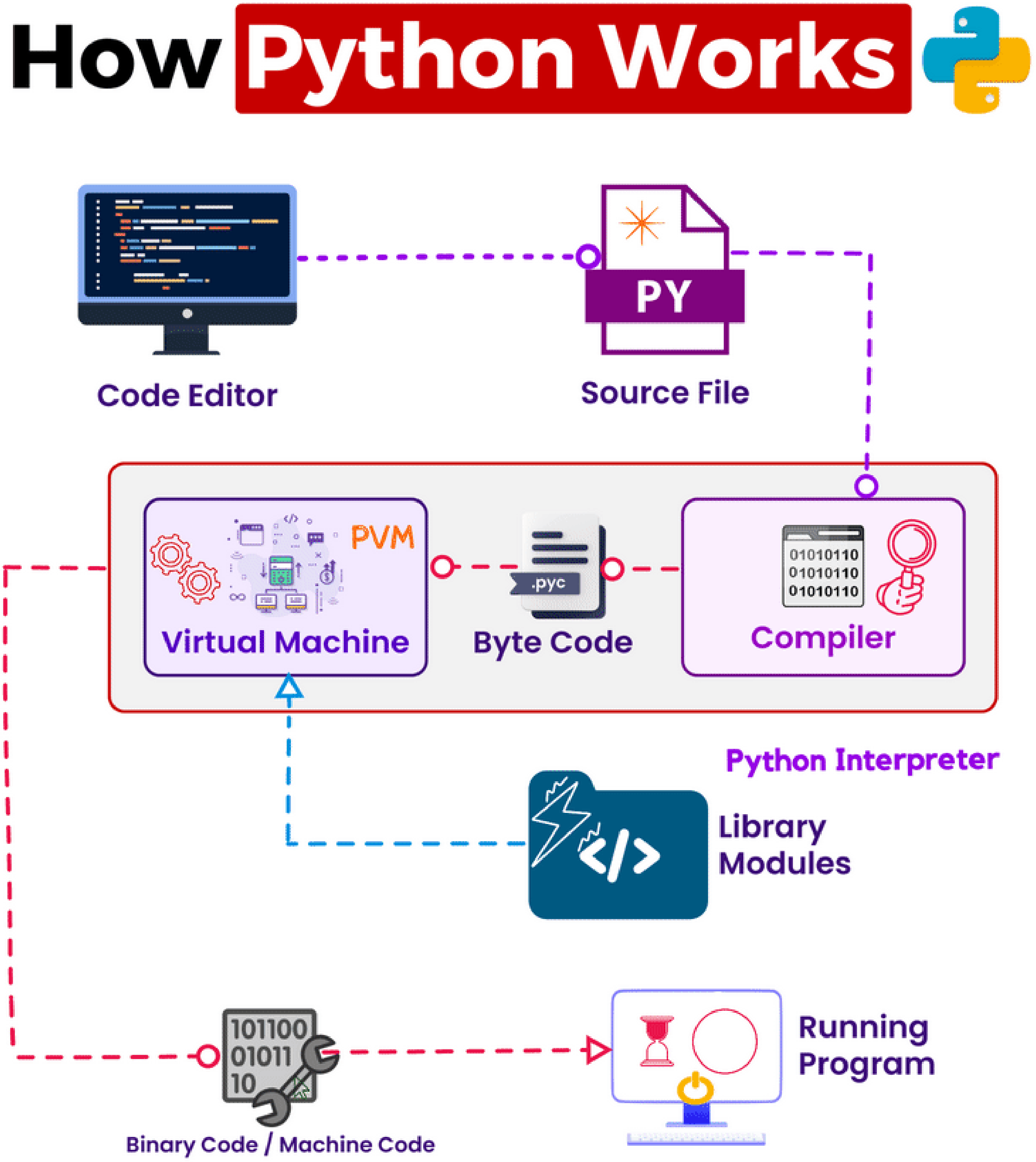
Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

=> Execution of Python program



Interpreter Vs Compiler

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.

Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Python Variables

Creating 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.

Example

```
x = 5
y = "Sanjeev"
print(x)
print(y)
```

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

Example

```
x = 4 # x is of type int
x = "Sally" # x is now of type str
print(x)
```

String variables can be declared either by using single or double quotes:

Example

```
x = "Sanjeev"
# is the same as
x = 'Sanjeev'
```

Variable Names

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

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assign Value to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Output Variables

The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the + character:

Example

```
x = "awesome"
print("Python is " + x)
```

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

For numbers, the + character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

If you try to combine a string and a number, Python will give you an error:

Example

```
x = 5
y = "Sanjeev"
print(x + y)
```

Python User Input

User Input

Python allows for user input.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

Python 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

Python 2.7

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

Python Data Types

Built-in Data Types

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`Str`
Numeric Types:	`int, float, complex`
Sequence Types:	`list, tuple, range`
Mapping Type:	`Dict`
Set Types:	`set, frozenset`
Boolean Type:	`Bool`
Binary Types:	`bytes, bytearray, memoryview`

Python Casting

Casting in python is therefore done using constructor functions:

- int( ) - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- float( ) - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str( ) - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

```
y = int(2.8) # y will be 2

z = float("3")  # z will be 3.0

y = str(2)    # y will be '2'
```

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example

Syntax Error:

```
if 5 > 2:
 print("Five is greater than two!")
      print("Five is greater than two!")
```

Python Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a `#`, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a # for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
""""
This is a comment
written in
more than just one line
""""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Operators

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

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

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

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

Comparison operators are used to compare two values:

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)

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Strings

String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example

Get the characters from position 5 to position 1, starting the count from the end of the string:

```
b = "Hello, World!"
print(b[-5:-2])
```

String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

String Methods

Python has a set of built-in methods that you can use on strings.

Example

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) #returns "Hello, World!"
```

Example

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Example

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) #returns ['Hello', ' World!']
```

Check String

To check if a certain phrase or character is present in a string, we can use the keywords `in` or `not in`.

Example

Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

Example

Check if the phrase "ain" is NOT present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable `a` with variable `b` into variable `c`:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is Sanjeev, I am " + age
print(txt)
```

But we can combine strings and numbers by using the `format()` method!

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

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is Sanjeev, and I am {}"
print(txt.format(age))
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

String Methods

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string

encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Python Lists

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

List

In Python lists are written with square brackets.

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

Access Items

You access the list items by referring to the index number:

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Negative Indexing

Negative indexing means beginning from the end

Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Note: The search will start at index 2 (included) and end at index 5 (not included).

- By leaving out the start value, the range will start at the first item:
- By leaving out the end value, the range will go on to the end of the list:
- Specify negative indexes if you want to start the search from the end of the list:

Change Item Value

Example

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

List Length

To determine how many items a list has, use the `len()` function:

Example

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Add Items

To add an item to the end of the list, use the append() method:

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

To add an item at the specified index, use the insert() method:

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Remove Item

Example

The `remove()` method removes the specified item:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Example

The `pop()` method removes the specified index, (or the last item if index is not specified):

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Example

The `del` keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

Example

The `del` keyword can also delete the list completely:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Example

The `clear()` method empties the list:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Copy a List

Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Join Two Lists

Example

Join two list:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

Example

Append list2 into list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)
```

```
print(list1)
```

Example

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

The list( ) Constructor

Example

Using the `list()` constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

List Methods

Python has a set of built-in methods that you can use on lists.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Python Tuples

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Negative Indexing

Negative indexing means beginning from the end

Example

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Range of Indexes

When specifying a range, the return value will be a new tuple with the specified items.

Example

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
```

```
print(x)
```

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the commma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Remove Items

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

Example

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Join Two Tuples

Example

Join two tuples:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

The tuple( ) Constructor

It is also possible to use the tuple() constructor to make a tuple.

Example

Using the tuple( ) method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Python Sets

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Access Items

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the `add()` method.

To add more than one item to a set use the `update()` method.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Example

Add multiple items to a set, using the `update()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])
```

```
print(thisset)
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

Note: Sets are *unordered*, so when using the `pop()` method, you will not know which item that gets removed.

Example

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

Example

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

Join Two Sets

There are several ways to join two or more sets in Python.

Example

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

Example

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

Note: Both `union()` and `update()` will exclude any duplicate items.

The set() Constructor

Example

Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Python Dictionaries

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

```
x = thisdict["model"]
```

There is also a method called `get()` that will give you the same result:

Example

```
x = thisdict.get("model")
```

Change Values

You can change the value of a specific item by referring to its key name:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Loop Through a Dictionary

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Example

Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():
  print(x, y)
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Example

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Example

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Copy a Dictionary

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.

Example

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to nest three dictionaries that already exists as dictionaries:

Example

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
```

```
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

The dict() Constructor

It is also possible to use the dict() constructor to make a new dictionary:

Example

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Python If ... Else

If

An "if statement" is written by using the if keyword.

Example

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Else

The else keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

```
a = 2
b = 330
print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

Example

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

Example

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
  pass
```

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

```
i = 1
while i < 6:
 print(i)
 i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

```
i = 1
while i < 6:
 print(i)
 if i == 3:
  break
 i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

```
i = 0
while i < 6:
 i += 1
 if i == 3:
  continue
 print(i)
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

Example

```
i = 1
while i < 6:
 print(i)
 i += 1
else:
 print("i is no longer less than 6")
```

For Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

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

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

```
for x in range(6):
  print(x)
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

range(2, 6), which means values from 2 to 6 (but not including 6):

Example

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example

```
for x in range(2, 30, 3):
  print(x)
```

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters or Arguments?

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

```
x = lambda a : a + 10
print(x(5))
```

Example

A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

```
# mymodule.py
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

```
import mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the **syntax:** *module_name.function_name*.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

```
import platform

x = platform.system()
print(x)
```