



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.
- Python will give you an error if you skip the indentation
- The number of spaces is up to you as a programmer, but it has to be at least one.
- You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

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.

How to use comments:

- Comments starts with a **#**, and Python will ignore them.
- Comments can be placed at the end of a line, and Python will ignore the rest of the line.
- A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code.

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

NOTE: To give multiline Comment, Use shortcut Key **ctrl+/.**

Multi Line Comments:

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

To add a multiline comments you could insert a **#** for each line.

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

Or, not quite as intended, you can use a multiline string.

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.

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

PYTHON VARIABLES:

- Variables are entities which help us store information and retrieve it later.
- Variables are containers for storing data values.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Variables Do not have any particular data type.
- You can change the value of variable in your program at any time, and python will always keep track of its current value.
- Variable names are case-sensitive.

Casting:

If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)      # x will be '3'
y = int(3)      # y will be 3
z = float(3)    # z will be 3.0
```

NOTE: For find the data type you can use the `type()` function.

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.
- Spaces are not allowed in variable names.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _). Names can not contain any of these symbols: `:"'<>/?|\\!@#%^&*~--+`
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Do not use Python Keywords and Function names like `list`, `str`, `def` etc.

Multi Words Variable Names:

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

1) Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

2) **Pascal Case**

Each word starts with a capital letter:

```
MyVariableName = "John"
```

3) **Snake Case**

Each word is separated by an underscore character:

```
my_variable_name = "John"
```

Assign Multiple Values:

Many Values to Multiple Variables:

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

```
x, y, z = "Orange", "Banana", "Cherry"
```

One Value to Multiple Variables:

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

```
x = y = z = "Orange"
```

Unpack a Collection:

If you have a collection of values in a list, tuple etc. Python allows you extract the values into variables. This is called *unpacking*.

```
fruits = ["apple", "banana", "cherry"]  
x, y, z = fruits
```

Output Variables:

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

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

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

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

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

NOTE: We can combine strings and numbers by using the `format()` method

Python Data types:

Built-in Data Types:

In programming, data type is an important concept.

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:	<code>str</code>
Numeric Types:	<code>int</code> , <code>float</code> , <code>complex</code>
Sequence Types:	<code>list</code> , <code>tuple</code> , <code>range</code>
Mapping Type:	<code>dict</code>
Set Types:	<code>set</code> , <code>frozenset</code>
Boolean Type:	<code>bool</code>
Binary Types:	<code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>

NOTE: You can get the data type of any object by using the `type()` function.

In Python, the data type is set when you assign a value to a variable.

```
x = "Hello World"
x = 20
x = 20.5
```

If you want to specify the data type, you can use the following constructor functions.

```
x = str("Hello World")
x = int(20)
x = float(20.5)
```

NOTE: You can convert any int, float into string but can't convert string into int, float.

Numeric Types:

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

1) int:

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522
```

2) Float:

Python calls any number with a decimal point a float.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (E) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
z = -35.59x = 35e3  
y = 12E4  
z = -87.7e100
```

3) Complex:

Complex numbers are written with a "j" as the imaginary part.

```
x = 3+5j  
y = 5j  
z = -5j
```

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods.

Python support the order of operations.

You can also use parentheses to modify the order of operations.

When you divide any two numbers, even if they are integers that result in a whole numbers, you'll always get a float.

If you mix an integer and a float in any other operation, you'll get a float.

NOTE: When you're writing long numbers, you can group digits using underscore to make large numbers more readable. When you print this numbers it print only the digits.

```
print(50_00_000)  
  
>>> 5000000
```

CONSTANT: Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed.

```
MAX_CONNECTIONS=5000
```

Python String:

String:

Strings are used in Python to record text information, such as names. It could either be a word, a phrase, a sentence, a paragraph or an entire encyclopedia. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "joker" to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

Creating a String:

To create a string in Python you need to use either single quotation marks, or double quotation and Triple quotes marks 'hello' is the same as "hello".
'''hello'''.

Strings are **immutable**.

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

```
print("Hello")  
  
>>> Hello
```

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

```
a = "Hello"
```

Multiline Strings

```
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod team"""
```

Or


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

NOTE: We can use string for looping

String Indexing:

- In python, individual characters of the string can be accessed by using the method of indexing.
- We know strings are a sequence, which means Python can use indexes to call parts of the sequence.
- A string index refers to the location of an element present in a string.
- The indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.
- In Python, we use brackets `[]` after an object to call its index.

G	E	E	K	S	F	O	R	G	E	E	K	S
0	1	2	3	4	5	6	7	8	9	10	11	12
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



```
string_1 = "Mrunal"

print("Second character of the string is:")
print(string_1[1])
```

```
>>> Second character of the string is:
r
```

String slicing:

- You can return a range of characters by using the slice syntax.
- Use a `:` to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the `:` and the ending index is specified on the right of the `:`
- Specify the start index and the end index, separated by a colon, to return a part of the string.

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

```
>>> llo
```

NOTE: Start Index is included but End Index is Non-Included.

Slice From the Start

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

```
>>> Hello
```

Slice To the End

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

```
>>> llo, World!
```

Entire String

If you do not specify the starting and the ending index, it will extract all elements of the string.

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

```
>>> Hello, World!
```

Or

```
b = "Hello, World!"  
print(b[0:])
```

```
>>> Hello, World!
```

Negative Indexing

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

In negative Indexing it start counting from Right Side.

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

```
>>> orl
```

We can also use index and slice notation to grab elements of a sequence by a specified step size

```
b = "Principal Component Analysis!"  
print(b[::1])  
-----
```

```
>>> Principal Component Analysis!
```

Grab Everything, but go in step size of 5.

```
b = "Principal Component Analysis!"  
print(b[5:15:5])  
-----
```

```
>>> iC
```

We can use this to print a string reverse.

```
b = "Principal Component Analysis!"  
print(b[::-1])  
-----
```

```
>>> !sisylanA tnenopmoC lapicnirP
```

String Concatenation:

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

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

```
>>> HelloWorld
```

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

```
>>> Hello World
```

Using * with string :

```
a = "Hello"
c = a*3
print(c)
-----
>>> HelloHelloHello
```

String Format:

In Python a string of required formatting can be achieved by different methods:

- 1)Using {}
- 2)Using %
- 3)Using f-string

1) Using {}:

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:

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

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

```
quantity = 3
item_no = 567
price = 49.95
my_order = "I want {} pieces of item {} for {} dollars."
print(my_order.format(quantity, item_no, price))
-----
>>> I want 3 pieces of item 567 for 49.95 dollars.
```

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

```
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))
```

```
>>> I want to pay 49.95 dollars for 3 pieces of item 567.
```

```
String1 = "{} {} {}".format('love', 'is', 'Life')
print("Print String in default order: ")
print(String1)
```

```
>>> Print String in default order:
      love is Life
```

Positional Formatting:

```
# Positional Formatting
String1 = "{1} {0} {2}".format('is', 'Love', 'Life')
print("\nPrint String in Positional order: ")
print(String1)
```

```
>>> Print String in order of Keywords:
      Love is Life
```

Keyword Formatting:

```
# Keyword Formatting
String1 = "{1} {f} {g}".format(g = 'Life', f = 'is', l = 'Love')
print("\nPrint String in order of Keywords: ")
print(String1)
```

```
>>>Print String in order of Keywords:
      Love is Life
```

Formatting of Intrgers:

```
# Formatting of Integers
String1 = "{0:b}".format(16)
print("\nBinary representation of 16 is ")
print(String1)
```

```
>>> Binary representation of 16 is
10000
```

Formatting of Floats:

```
# Formatting of Floats
String1 = "{0:e}".format(165.6458)
print("\nExponent representation of 165.6458 is ")
print(String1)
```

```
>>> Exponent representation of 165.6458 is
1.656458e+02
```

Rounding of Integers:

```
# Rounding off Integers
String1 = "{0:.2f}".format(1/6)
print("\none-sixth is : ")
print(String1)
-----
>>> one-sixth is :
      0.17
```

2)Using %:

```
# Initialize variable as a string
variable = '15'
string = "Variable as string = %s" % (variable)
print(string)

# Printing as raw data
print("Variable as raw data = %r" % (variable))

# Convert the variable to integer
# And perform check other formatting options

variable = int(variable)
# Without this the below statement will give error.
string = "Variable as integer = %d" % (variable)
print(string)
print("Variable as float = %f" % (variable))

# printing as any string or char after a mark
# here i use mayank as a string
print("Variable as printing with special char = %cmayank" % (variable))

print("Variable as hexadecimal = %x" % (variable))
print("Variable as octal = %o" % (variable))
-----
>>> Variable as string = 15
      Variable as raw data = '15'
      Variable as integer = 15
      Variable as float = 15.000000
      Variable as printing with special char = mayank
      Variable as hexadecimal = f
      Variable as octal = 17
```

f-string:

Now a day's **f-string** is mostly use. In some situation, you'll want to use a variable's value inside a string.

```
first_name = 'Mrunal'
last_name = 'Wankhede'
full_name = f'{first_name} {last_name}'
print(full_name)
-----
>>> Mrunal Wankhede
```

Escape Character:

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a backslash `\` followed by the character you want to insert.
- An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
# You will get an error if you use double quotes inside a string that is
surrounded by double quotes:
txt = "We are the so-called "Vikings" from the north."
-----

>>> SyntaxError: invalid syntax
```

To fix this problem, use the escape character `\`:

```
# The escape character allows you to use double quotes when you normally
would not be allowed:

txt = "We are the so-called \"Vikings\" from the north."
print(txt)
-----

>>> We are the so-called "Vikings" from the north.
```

Other escape characters used in Python:

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

NOTE: applying any method on a string does not actually change the original string. We need to store the modified string in another variable.

String functions and methods:

Method	Description
<code>capitalize()</code>	Converts the first character to upper case
<code>casefold()</code>	Converts string into lower case
<code>center()</code>	Returns a centered string
<code>count()</code>	Returns the number of times a specified value occurs in a string
<code>encode()</code>	Returns an encoded version of the string
<code>endswith()</code>	Returns true if the string ends with the specified value
<code>expandtabs()</code>	Sets the tab size of the string
<code>find()</code>	Searches the string for a specified value and returns the position of where it was found
<code>format()</code>	Formats specified values in a string
<code>format_map()</code>	Formats specified values in a string
<code>index()</code>	Searches the string for a specified value and returns the position of where it was found
<code>isalnum()</code>	Returns True if all characters in the string are alphanumeric
<code>isalpha()</code>	Returns True if all characters in the string are in the alphabet
<code>isascii()</code>	Returns True if all characters in the string are ascii characters
<code>isdecimal()</code>	Returns True if all characters in the string are decimals
<code>isdigit()</code>	Returns True if all characters in the string are digits
<code>isidentifier()</code>	Returns True if the string is an identifier
<code>islower()</code>	Returns True if all characters in the string are lower case
<code>isnumeric()</code>	Returns True if all characters in the string are numeric
<code>isprintable()</code>	Returns True if all characters in the string are printable
<code>isspace()</code>	Returns True if all characters in the string are whitespaces
<code>istitle()</code>	Returns True if the string follows the rules of a title
<code>isupper()</code>	Returns True if all characters in the string are upper case
<code>join()</code>	Converts the elements of an iterable into a string
<code>ljust()</code>	Returns a left justified version of the string
<code>lower()</code>	Converts a string into lower case
<code>lstrip()</code>	Returns a left trim version of the string
<code>maketrans()</code>	Returns a translation table to be used in translations
<code>partition()</code>	Returns a tuple where the string is parted into three parts
<code>replace()</code>	Returns a string where a specified value is replaced with a specified value
<code>rfind()</code>	Searches the string for a specified value and returns the last position of where it was found

<code>rindex()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rjust()</code>	Returns a right justified version of the string
<code>rpartition()</code>	Returns a tuple where the string is parted into three parts
<code>rsplit()</code>	Splits the string at the specified separator, and returns a list
<code>rstrip()</code>	Returns a right trim version of the string
<code>split()</code>	Splits the string at the specified separator, and returns a list
<code>splitlines()</code>	Splits the string at line breaks and returns a list
<code>startswith()</code>	Returns true if the string starts with the specified value
<code>strip()</code>	Returns a trimmed version of the string
<code>swapcase()</code>	Swaps cases, lower case becomes upper case and vice versa
<code>title()</code>	Converts the first character of each word to upper case
<code>translate()</code>	Returns a translated string
<code>upper()</code>	Converts a string into upper case
<code>zfill()</code>	Fills the string with a specified number of 0 values at the beginning

Python Booleans:

- Booleans represent one of two values: **True** or **False**.
- In programming you often need to know if an expression is **True** or **False**.
- You can evaluate any expression in Python, and get one of two answers, **True** or **False**.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer
- The **bool()** function allows you to evaluate any value, and give you **True** or **False** in return,

```
print(bool("Hello"))
print(bool(15))
```

```
>>> True
      True
```

Almost any value is evaluated to **True** if it has some sort of content.

Any string is **True**, except empty strings.

Any number is **True**, except **0**.

Any list, tuple, set, and dictionary are **True**, except empty ones.

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

In fact, there are not many values that evaluate to **False**, except empty values, such as **()**, **[]**, **{}**, **""**, the number **0**, and the value **None**. And of course the value **False** evaluates to **False**.

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

One more value, or object in this case, evaluates to **False**, and that is if you have an object that is made from a class with a `__len__` function that returns **0** or **False**:

```
class myclass():
    def __len__(self):
        return 0

myobj = myclass()
print(bool(myobj))
-----

>>> False
```

This class topic will be explain later in Notes.

Functions can Return a Boolean:

You can create functions that returns a Boolean Value:

```
# Print the answer of a function:
def myFunction() :
    return True

print(myFunction())
-----

>>> True
```

You can execute code based on the Boolean answer of a function:

```
# Print "YES!" if the function returns True, otherwise print "NO!":
def myFunction() :
    return True

if myFunction():
    print("YES!")
else:
    print("NO!")
-----

>>> YES!
```

Python also has many built-in functions that return a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

```
# Check if an object is an integer or not:
x = 200
print(isinstance(x, int))
-----

>>> True
```

Check if a string contains a particular word or character

```
my_string = 'Albert Einstein'
print('Albert' in my_string)
print('Alberta' in my_string)
-----

>>> True

      False
```

Python Operators:

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators (+, -, *, /, %, **, //)
- Assignment operators (=, +=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, v, <<=)
- Comparison operators (==, !=, >, <, >=, <=)
- Logical operators (and, or, not)
- Identity operators (is, is not)
- Membership operators (in, not in)

- Bitwise operators (&, |, ^, ~, <<, >>)

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	$x // y$

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

Comparison Operators

Comparison operators are used to compare two values:

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

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)

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

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

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

The operator precedence in Python is listed in the following table. It is in descending order (upper group has higher precedence than the lower ones).

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT

and	Logical AND
or	Logical OR

Python Lists:

List:

- Lists are used to store multiple items in a single variable.
- Lists can be thought of the most general version of a sequence in Python.
- Unlike strings, they are mutable, meaning the elements inside a list can be changed!
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.
- Lists are constructed with brackets `[]` and commas separating every element in the list

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

>>> ['apple', 'banana', 'cherry']
<class 'list'>
```

List Items:

- List items are ordered, changeable, and allow duplicate values. You can add in list int, float, string, Tuple, Dictionary, etc.

Ordered:

- When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
- If you add new items to a list, the new items will be placed at the end of the list.

Note: There are some list methods that will change the order, but in general: the order of the items will not change.

Changeable:

- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates:

- Since lists are indexed, lists can have items with the same value:
- List items can be of any data type:

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

The list() Constructor:

It is also possible to use the `list()` constructor when creating a new list.

```
# Using the list() constructor to make a List:
thislist = list(("apple", "banana", "cherry")) # note the double round-
brackets
print(thislist)
-----
```

```
>>>['apple', 'banana', 'cherry']
```

List Indexing:

Access Items:

- Indexing work just like in strings.
- A list index refers to the location of an element in a list.
- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

```
# Print the second item of the list:
thislist = ["apple", "banana", "cherry"]
print(thislist[1])

>>> banana
```

Negative Indexing:

Negative indexing means start from the end.

-1 refers to the last item, **-2** refers to the second last item etc.

```
# Print the last item of the list:
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
-----

>>> cherry
```

List Slicing:

- We can use a **:** to perform slicing which grabs everything up to a designated point.
- The starting index is specified on the left of the **:** and the ending index is specified on the right of the **:**
- Remember the element located at the right index is not included.

```
# Return the third, fourth, and fifth item:
thislist=["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
-----
```

```
>>> ['cherry', 'orange', 'kiwi']
```

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:

```
# This example returns the items from the beginning to, but NOT including,
"kiwi":
thislist=["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
-----

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

By leaving out the end value, the range will go on to the end of the list:

```
# This example returns the items from "cherry" to the end:
thislist=["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
-----

>>> ['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

If you do not specify the starting and the ending index, it will extract all elements of the list.

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

>>> ['apple', 'banana', 'cherry', 'orange', 'kiwi', 'melon', 'mango']
```

Range of Negative Indexes:

Specify negative indexes if you want to start the search from the end of the list:

```
# This example returns the items from "orange" (-4) to, but NOT including
"mango" (-1):
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
-----

>>> ['orange', 'kiwi', 'melon']
```

Change List Items:

We can also use + to concatenate lists, just like we did for strings

```
thislist = ["apple", "banana", "cherry"]
add_thislist = thislist + ['Mango']
print(add_thislist)
-----
>>> ['apple', 'banana', 'cherry', 'Mango']
```

Change Item Value:

To change the value of a specific item, refer to the index number:

```
# Change the second item:
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
-----
>>> ['apple', 'blackcurrant', 'cherry']
```

Change a Range of Item Values:

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

```
# Change the values "banana" and "cherry" with the values "blackcurrant"
and "watermelon":
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
-----
>>> ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

```
# Change the second value by replacing it with two new values:
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
-----
>>> ['apple', 'blackcurrant', 'watermelon', 'cherry']
```

NOTE: The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

```
# Change the second and third value by replacing it with one value:
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)

-----

>>> ['apple', 'watermelon']
```

Nested List:

A great feature of Python data structures is that they support nesting. This means we can have data structures within data structures.

For example: A list inside a list.

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=['b','a','d']
lst_3=[7,8,9]

# Make a list of lists to form a matrix
list_of_lists = [lst_1,lst_2,lst_3]
print(list_of_lists)

# Grab first item in matrix object
print(list_of_lists[1])

-----

>>>[[1, 2, 3], ['b', 'a', 'd'], [7, 8, 9]]
     ['b', 'a', 'd']
```

Python List/Array Methods:

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

Python Tuples:

Tuple:

- Tuples are used to store multiple items in a single variable.
- Tuple items can be of any data type. A tuple can contain different data types.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

- A tuple is a collection which is ordered and **unchangeable**.
- You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

Constructing Tuples:

The construction of a tuples use `()` with elements separated by commas.

```
thistuple = ("apple", "banana", "cherry")
thistuple = "apple", "banana", "cherry"    # This is also Tuple.
my_tuple = (1, 'a', 3)
another_tuple = ('one', 2, 4.53, 'asbc')
my_tuple = (1,)
```

Tuple Items:

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered:

- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates:

- Since tuples are indexed, they can have items with the same value:

NOTE: 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.

The tuple() Constructor:

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

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

Tuple Indexing:

- Indexing work just like in lists.
- A tuple index refers to the location of an element in a tuple.
- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

Access Tuple Items:

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

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

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, **-2** refers to the second last item etc.

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

Tuple Slicing:

- We can use a **:** to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the **:** and the ending index is specified on the right of the **:**
- Remember the element located at the right index is not included.

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

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:

```
thistuple=("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
-----
>>> ('apple', 'banana', 'cherry', 'orange')
```

By leaving out the end value, the range will go on to the end of the list:

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

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

```
thistuple=("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
-----
>>> ('orange', 'kiwi', 'melon')
```

Update Tuples:

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

Change Tuple Values:

You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Add Items:

Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. **Convert into a list:** Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

2. **Add tuple to a tuple.** You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
-----
>>> ('apple', 'banana', 'cherry', 'orange')
```

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

- 1) Convert the tuple into a list, remove "apple", and convert it back into a tuple.
- 2) You can delete the tuple completely.

Unpack Tuples:

Unpacking a Tuple:

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

```
fruits = ("apple", "banana", "cherry")
```

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Unpacking a Tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
-----
>>> apple
```

```
banana
cherry
```

NOTE: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk*:

If the number of variables is less than the number of values, you can add an `*` to the variable name and the values will be assigned to the variable as a list:

```
# Assign the rest of the values as a list called "red":
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
-----

>>> apple
      banana
      ['cherry', 'strawberry', 'raspberry']
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

```
# Add a list of values the "tropic" variable:
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
-----

>>> apple
      ['mango', 'papaya', 'pineapple']
      cherry
```

Join Tuples:

Join Two Tuples:

To join two or more tuples you can use the **+** operator:

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

tuple3 = tuple1 + tuple2
print(tuple3)
-----
>>> ('a', 'b', 'c', 1, 2, 3)
```

Multiply Tuples:

If you want to multiply the content of a tuple a given number of times, you can use the ***** operator:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
-----
>>> ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

When to use Tuples:

- You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.
- You will find them often in functions when you are returning some values
- You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Tuples Method:

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

Python Sets:

- Sets are an unordered collection of **unique elements**. We can construct them by using the `set()` function.

```
empty_set = set()
```

- An empty set cannot be represented as `{}`, which is reserved for an empty dictionary which we will get to know in a short while
- Sets cannot have duplicates.
- Sets are mutable just like lists.
- You can create a non-empty set with curly braces by specifying elements separated by a comma.

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

Set:

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Tuple](#), and [Dictionary](#), all with different qualities and usage.
- A set is a collection which is both **unordered** and **unindexed**.

- Sets are written with curly brackets.

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items:

- Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered:

- Unordered means that the items in a set do not have a defined order.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable:

- Sets are unchangeable, meaning that we cannot change the items after the set has been created.

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

Duplicates Not Allowed:

- Sets cannot have two items with the same value.

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

Set Items - Data Types:

- A set can contain different data types:

```
data_types_set = {"apple", 23, 1.25, True}
print(data_types_set)
-----
>>> {1.25, 'apple', True, 23}
```

We cannot create a set whose any of the elements is a list. But we can have tuples as set elements, they are immutable.


```
my_set = {1, 2, (2,3)}  
print(my_set)
```

```
>>> {(2, 3), 1, 2}
```

The set() Constructor:

- It is also possible to use the `set()` constructor to make a set.
- Using the `set()` constructor to make a set:

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

```
>>> {'banana', 'apple', 'cherry'}
```

Access Set Items:

Access Items:

- You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Change Items:

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

Some functions you can use in Set:

- (1) `intersection()`
- (2) `union()`

```
set_U = {1,2,3,4,5,6,7,8,9,10}
```

```
A = {1,2,3,4,5}
```

```
B = {2,4,6,8,10}
```

```
C = {1,3,5,7,9}
```

```
output_1 = A.intersection(B)
```

```
print(output_1)
```

```

output_2 = A.union(B)
print(output_2)

output_3 = B.union(C)
print(output_3)

output_4 = B.intersection(C)
print(output_4)

output_5 = A.intersection(B).intersection(C)
print(output_5)

>>> {2, 4}
      {1, 2, 3, 4, 5, 6, 8, 10}
      {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
      set()
      set()

```

Set Methods:

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

Python Dictionaries:

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

Dictionary:

- We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python.
- If you're familiar with other languages you can think of these Dictionaries as hash tables.
- So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.
- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values.

Dictionary Items:

- Dictionary items are unordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Unordered:

- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Changeable:

- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed:

- Dictionaries cannot have two items with the same key

Dictionary Items - Data Types:

- The values in dictionary items can be of any data type.

Constructing a Dictionary:

- A dictionary object is constructed using curly braces

```
{key1:value1, key2:value2, key3:value3}
```

Access Dictionary Items:

Accessing Items:

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

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

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

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

NOTE: There is also method like `get()`, `keys()`, `values()`, `items()`.

Change Dictionary Items:

Change Values:

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

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
}
```

```
    "year": 1964
}
thisdict["year"] = 2018
print(thisdict)
-----
>>> {'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Update Dictionary:

The `update()` method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict.update({"year": 2020})
print(thisdict)
-----
>>> {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Add Dictionary Item:

Adding Items:

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

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

Update Dictionary:

The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})  
print(thisdict)  
-----  
>>> {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Remove Dictionary Items:

`pop()`

`popitem()`

`del`

`clear()`

Python - Copy Dictionaries:

There are some Methods:

`Copy()`

`Dict()` – To make a new dict

Nested Dictionaries:

Nested Dictionaries:

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

```
myfamily = {  
    "child1" : {  
        "name" : "Emil",  
        "year" : 2004  
    },  
}
```

```
"child2" : {
    "name" : "Tobias",
    "year" : 2007
},
"child3" : {
    "name" : "Linus",
    "year" : 2011
}
}
```

Dictionaries Methods:

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

Python If ... Else:

Python supports the usual logical conditions from mathematics:

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

These conditions can be used in several ways, most commonly in "if statements" and loops.

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:"Hey if this case happens, perform some action"

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

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

```
>>> 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.

Elif

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

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

```
>>> a and b are equal
```

Else

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

```
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")
-----
```

```
>>> a is greater than b
```


You can also have an **else** without the **elif**:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
-----
```

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

Short Hand If

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

```
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:

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

```
>>> B
```

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

```
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:

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

```
>>> Both conditions are True
```

Or

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

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

```
>>> At least one of the conditions is True
```

Nested If

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

```
x = 41

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

```
>>> Above ten,
    and also 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.

```
a = 33
b = 200

if b > a:
    pass
```

Python While Loops:

Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

The while statement in Python is one of the most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

The 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.

The while statement in Python is one of the most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

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

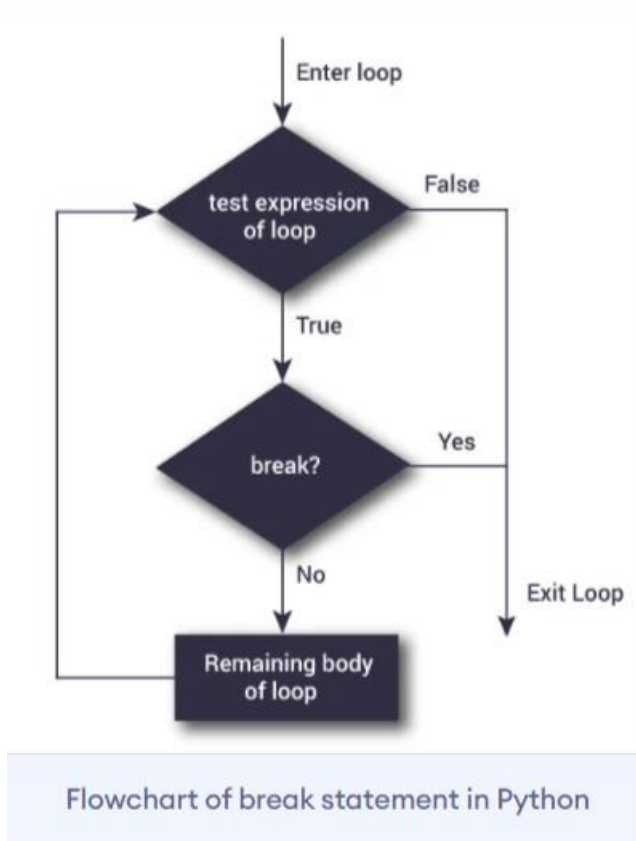
```
i = 1
while i < 6:
    print(i)
    i += 1
```

NOTE: Remember to increment i, or else the loop will continue forever.

The **while** loop requires relevant variables to be ready, in this example we need to define an indexing variable, **i**, which we set to 1.

The break Statement:

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

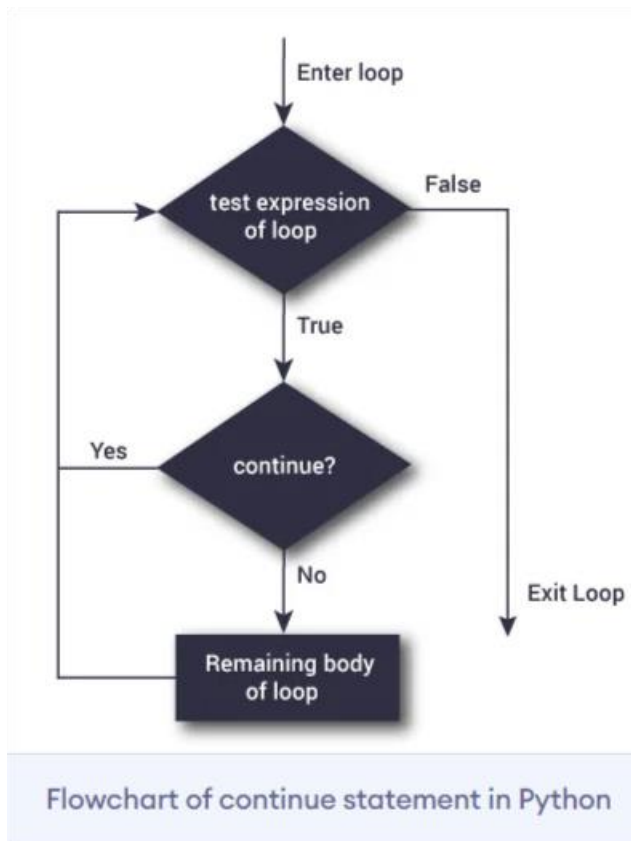


```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1

>>> 1
      2
      3
```

The continue Statement:

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



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

```
>>> 1
      2
      4
      5
      6
```

The else Statement:

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

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

```
>>> 1
      2
      3
      4
```

```
5
i is no longer less than 6
```

Python 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.

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

```
>>> apple
      banana
      cherry
```

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:

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

```
>>> b
      a
      n
      a
      n
      a
```

Looping Through a List:

```
# A simple for loop to print the houses of GOT universe
got_houses = ['Stark', 'Arryn', 'Baratheon', 'Tully', 'Greyjoy',
              'Lannister', 'Tyrell', 'Martell', 'Targaryen']
for house in got_houses[::-1]:
    print(f"House {house}")
-----

>>> House Targaryen
      House Martell
      House Tyrell
      House Lannister
      House Greyjoy
      House Tully
      House Baratheon
      House Arryn
      House Stark
```

EXAMPLE: Suppose you are given a list of numbers. You need to find the corresponding squares of these numbers and zip them together in a dictionary

```
# The list of numbers
list_of_numbers = [1, 2, 4, 6, 11, 14, 17, 20]
# Let us first print the squares
for number in list_of_numbers:
    squared_number = number**2
    print(f"The square of {number} is {squared_number}")
-----

>>> The square of 1 is 1
      The square of 2 is 4
      The square of 4 is 16
      The square of 6 is 36
      The square of 11 is 121
      The square of 14 is 196
      The square of 17 is 289
      The square of 20 is 400
```

range() Function:

range() function is a pretty useful function to get a sequence of numbers. It takes three arguments : (start, stop, step)

```
list(range(10))
-----

>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Another Example:


```
# Let us first print the squares
for number in range(0,10,2):
    squared_number = number**2
    print(f"The square of {number} is {squared_number}")
```

```
>>> The square of 0 is 0
      The square of 2 is 4
      The square of 4 is 16
      The square of 6 is 36
      The square of 8 is 64
```

We can also iterate through each element of tuple as well

```
# Suppose we have a tuple of days
days = ('Monday' , 'Tuesday' , 'Wednesday' , 'Thursday'
        , 'Friday', 'Saturday', 'Sunday')

for day in days:
    print(f"Today is {day}")
```

```
>>> Today is Monday
      Today is Tuesday
      Today is Wednesday
      Today is Thursday
      Today is Friday
      Today is Saturday
      Today is Sunday
```

NOTE: We can also Iterate for Dictionary.

The break Statement in for loop:

With the **break** statement we can stop the loop before it has looped through all the items:

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

```
>>> apple
      banana
      apple
```

The continue Statement in for loop:

```
for i in range(9):
    if i == 3:
        continue
    print(i)
```

```
>>> 0
      1
      2
      4
      5
      6
      7
      8
```

The range() Function in for loop:

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.

```
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.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

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

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

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

Else in For Loop

The **else** keyword in a **for** loop specifies a block of code to be executed when the loop is finished:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

```
>>> 0  
      1  
      2  
      3  
      4  
      5  
      Finally finished!
```

Note: The **else** block will NOT be executed if the loop is stopped by a **break** statement.

```
for x in range(6):  
    if x == 3: break  
    print(x)  
else:  
    print("Finally finished!")
```

```
>>> 0  
      1  
      2
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]  
  
for x in adj:  
    for y in fruits:  
        print(x, y)
```

```
>>> red apple  
      red banana  
      red cherry  
      big apple  
      big banana
```

```
big cherry
tasty apple
tasty banana
tasty cherry
```

The pass Statement

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

```
for x in [0, 1, 2]:
    pass
```

List comprehensions:

List comprehensions are a tool for transforming one list (any iterable actually) into another list. During this transformation, elements can be conditionally included in the new list and each element can be transformed as needed.

List comprehensions in Python are great, but mastering them can be tricky because they don't solve a new problem: they just provide a new syntax to solve an existing problem.

From loops to comprehensions:

Every list comprehension can be rewritten as a **for** loop but not every for loop can be rewritten as a list comprehension. The key to understanding when to use list comprehensions is to practice identifying problems that smell like list comprehensions. If you can rewrite your code to look just like this **for** loop, you can also rewrite it as a list comprehension:

```
'''
Let us take an example of getting the squares of the numbers in a list
'''
# The list of numbers
list_of_numbers = [1, 2, 4, 6, 11, 14, 17, 20]

# The usual way to do this using for loop is described below:
# Let us first initialize a list where we will be appending the squares in
each iteration

squared_numbers = []

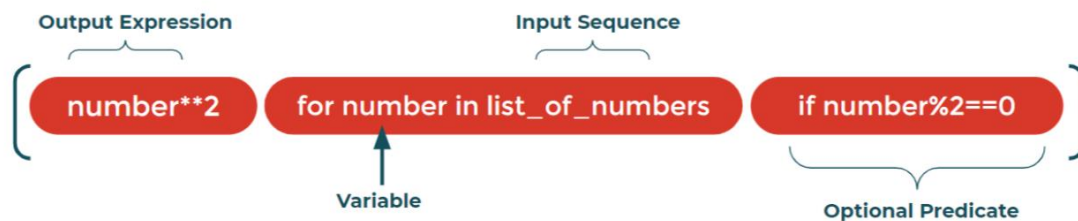
for number in list_of_numbers:
    # Use the append method to add the numbers one by one to our list
    squared_numbers.append(number**2)
    # print(squared_numbers)
```

```
print(f"The list of squared numbers is {squared_numbers}")

## We can do the same task using a list comprehension
# Getting the list of squares using list comprehension
squared_numbers = [number**2 for number in list_of_numbers]
print(squared_numbers)
```

A list comprehension consists of the following parts:

- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.



- The iterator part iterates through each member number of the input sequence `list_of_numbers`.
- The predicate checks if the member is even.
- If the member is even then it is passed to the output expression, squared, to become a member of the output list.

Python User Input:

User Input:

- Python allows for user input.
- That means we are able to ask the user for input.
- Python 3.6 uses the `input()` method.
- The following example asks for the username, and when you entered the username, it gets printed on the screen:

```
username = 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 Functions:

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

Blocks of code that are designed to do one specific Job. If you need to perform particular task again and again(multiple times) throughout your program, you don't need to type all the code for the same task again and again; you just call the function.

Creating a Function

In Python a function is defined using the `def` keyword

Let's see how to build out a function's syntax in Python. It has the following form:

```
def example_function(arg1, arg2):  
    '''
```

```
This is where the function's Document String (docstring) goes
'''
# Do stuff here
return desired_result
```

- We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example `len()` is a good name for a `length()` function. Also be careful with names, you wouldn't want to call a function the same name as a [built-in function in Python](#) (such as `len`).
- Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.
- Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programming languages do not do this, so keep that in mind.
- Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.
- After all this you begin writing the code you wish to execute.
- The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.
- `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Let's code:

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

In line 1 there is keyword **def** to inform python that you're defining the function. This is the **function definition**. At the end the parentheses is required. You can add information in those parentheses. Above code did not give any output. For that you have to call that function.

Calling a Function

If you want to run that block of code you have to call that function. To call a function, use the function name followed by parenthesis:

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

Passing information to a function:

If you want to add some information you can provide into the parentheses.

```
def greet_user(username):  
    print(f'Hello, {username.title()}')  
  
greet_user('mrunal')  
-----  
>>> Hello, Mrunal
```

Arguments and Parameters:

Arguments:

An argument is the value that is sent to the function when it is called. An argument is a piece of information that passed from a function call to function. The value 'mrunal' in greet_user('mrunal') is the example of **Arguments**.

Parameters:

A parameter is the variable listed inside the parentheses in the function definition. The variable username in the definition of greet_user is an example of a **parameter**.

Passing 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.

You can provide multiple parameters to function definition as you want, a function call may need multiple arguments.

You can pass arguments in number of ways:

POSITIONAL ARGUMENTS: It needs to be the parameters and arguments in same order.

KEYWORD ARGUMENTS: Each arguments consists of variable name and a value. (also list dictionary value)

Positional Arguments:

In positional arguments you have to maintain the position and orders of parameters and orders that they can match with each other's.

```
def describe_pet(animal_type, pet_name):  
    print(f'\nI have a {animal_type}')  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
describe_pet('hamster', 'harry')  
-----  
  
>>> I have a hamster  
      My hamster's name is Harry.
```

Multiple function calls:

You can call function as many times as you need.

```
def describe_pet(animal_type, pet_name):  
    print(f'\nI have a {animal_type}')  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
describe_pet('hamster', 'harry')  
describe_pet('dog', 'facebook')  
-----  
  
>>> I have a hamster  
      My hamster's name is Harry.  
  
      I have a dog  
      My dog's name is Facebook.
```

Orders matters in positonal arguments:

IF you mix up the orders of the arguments in a function call when using the position arguments you can get unexpected result.

```
def describe_pet(animal_type, pet_name):
    print(f"\nI have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
-----

>>> I have a harry
      My harry's name is Hamster.
```

Keyword Arguments:

You can also send arguments with the *key = value* syntax.

A keyword arguments is name-value pair that you type in function call and to pass the function definition, so there is no confusion to which value in assign in which name. They clarify the role of each value in the function call.

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
-----

>>> The youngest child is Linus
```

NOTE: When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

Default Values:

When writing a function, you can define a default value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value.

Both function call gives the same output

```
def my_function(country = "Norway"):
    print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
-----

>>> I am from Sweden
      I am from India
      I am from Norway
      I am from Brazil
```

Passing an arbitrary number of arguments:

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def make_pizza(*toppings):  
    print(toppings)  
  
make_pizza("pepperoni")  
my_function("mushrooms", "green peppers", "extra cheese")
```

Mixing Positional and Arbitrary Arguments:

Python matches positional and keywords arguments first and then collects any remaining arguments in the final parameter.

```
def function_name(*parameter):
```

Using Arbitrary Keyword Arguments:

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.

Return Value:

The value the function returns is called a return value. The return statement takes a value from inside a function and sends it back to the line that called the function.

```
def my_function(x):  
    return 5 * x  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))  
-----
```

```
>>> 15  
      25
```

NOTE: In python you can return Multiple values.

Making an Argument Optional:

Takes Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to.

```
def get_formatted_name(first_name, last_name, middle_name=''):
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
-----

>>> Jimi Hendrix
None
```

Passing a list:

Sometimes You can pass a list to a function. (list of names, numbers, float, dictionaries etc.)

```
def greet_user(names):
    for name in names:
        msg = f"Hello, {name.title()}"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_user(usernames)
-----

>>> Hello, Hannah
      Hello, Ty
      Hello, Margot
```

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`:

Save this code in a file named `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:

```
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):

```
person1 = {  
    "name": "John",  
    "age": 36,
```

```
    "country": "Norway"  
}
```

import the module named mymodule, and access the person1 dictionary:

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

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:

Create an alias for `mymodule` called `mx`:

```
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.

Import and use the `platform` module:

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

Using the dir() Function:

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The `dir()` function can be used on *all* modules, also the ones you create yourself.

Import From Module:

You can choose to import only parts from a module, by using the `from` keyword.

```
def greeting(name):
    print("Hello, " + name)

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module.

Example: `person1["age"]`, **not** `mymodule.person1["age"]`

Python Recursion:

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

If the recursion program goes into infinite times it runs only 1000 Times.

But if You want to run this program as our requirement you can use `sys.setrecursionlimit(2000)`

```
def tri_recursion(k):
    if(k>0):
        result = k+tri_recursion(k-1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)

>>> Recursion Example Results
1
3
6
10
15
21
```

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.

2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
 2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
 3. Recursive functions are hard to debug.
-

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:

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

>>> 15
```

Lambda functions can take any number of arguments:

Multiply argument **a** with argument **b** and return the result:

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

>>> 30
```

Summarize argument **a**, **b**, and **c** and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))

>>> 13
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

```
def myfunc(n):
    return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

```
def myfunc(n):
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Classes and Objects:

Python Classes/Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

```
class MyClass:
    x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

```
class MyClass:
    x = 5
p1 = MyClass()
print(p1.x)
```

The `__init__()` Function

- The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
- To understand the meaning of classes we have to understand the built-in `__init__()` function.
- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)
# p1 is Instance, Object.
# "John",36 is Attributes.

print(p1.name)
print(p1.age)

>>> John
      36
```

NOTE: The `__init__()` function is called automatically every time the class is being to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        # This is the Object Method
        print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)          # Making Object
p1.myfunc()                     # Calling Function

>>> Hello my name is John
```

NOTE: The **self** parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

Creating Multiple Instances:

You can create as many instances from a class as you need.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):              # This is the Object Method
        print("Hello my name is " + self.name)

p1 = Person("John", 36)          # Making Object
p2 = Person('Mrunal', 22)       # Making multiple Object, Instance

p1.myfunc()                     # Calling Function
p2.myfunc()

>>> Hello my name is John
      Hello my name is Mrunal
```

Setting a Default Value for an Attributes:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

        self.sex = "Male"       # Set default Value

    def myfunc(self):
        print("Hello my name is " + self.name)

    def abouts(self):            # Make an Method for an default value
```

```
print(f" The sex of person p1 is {self.sex}")

p1 = Person("John", 36)
p2 = Person("Mrunal", 22)

p1.abouts()                # Calling Methods
p1.myfunc()
p2.myfunc()
```

Modify Attribute Values:

You can change an attributes value in three ways:

- 1) Modifying an Attributes value Directly
- 2) Modifying an Attributes value Through a Method
- 3) Incrementing an Attribute value Through a Method

1. Modifying an Attributes value Directly:

```
class Computer:

    def __init__(self):
        self.name = "Navin"
        self.age = 28

c1 = Computer()
c2 = Computer()

c1.name = "Rashi"          # Changing the values
c2.age = 12

print (c1.name, c1.age)
print (c2.name, c2.age)
```

2. Modifying an Attributes value Through a Method

```
class Computer:

    def __init__(self):
        self.name = "Navin"
        self.age = 28

    def update (self):          # Make an Method for changing value
        self.age = 30

    c1 = Computer()
    c2 = Computer()

    c1.update()               # Calling Method

    print (c1.name, c1.age)
    print (c2.name, c2.age)
```

Compare two Object:

```
class Computer:

    def __init__(self):
        self.name = "Navin"
        self.age = 28

    def compare (self,other):    # c1 = self,    c2 = ohter
```

```
    if self.age == other.age:
        return True
    else:
        return false

c1 = Computer()
c2 = Computer()

c1.age = 30

if c1.compare(c2):
    print("They are Same")
else:
    print("They are different")

print (c1.name, c1.age)
print (c2.name, c2.age)
```

Types of Variables:

- 1) Instance Variable (For Changing values for Specific Object)
- 2) Class (Static) variable (It affect on all the Object)

1. Instance variables:

```
class Car:

    def __init__(self):
        self.mil = 10          # Instance Variable
        self.com = "BMW"

c1 = Car()
```



```
c2 = Car()

c1.mil = 8          # Instance Variable

print (c1.com, c1.mil)
print (c2.com, c2.mil)
```

2. Class Variables:

```
class Car:

    wheels = 4          # Class Variables

    def __init__(self):
        self.mil = 10
        self.com = "BMW"

c1 = Car()
c2 = Car()

c1.wheels = 8

print (c1.com, c1.mil, c1.wheels)
print (c2.com, c2.mil, c1.wheels)
```

Types of Methods:

1) Instance Method:

1. Accessor Method (For asseccking the value)
2. Mutator Methods (For modifying the values)

2) Class Method

3) Static Method

1. Instance Methods:

```
class Student:

    school = "Telusko"

    def __init__(self):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    def avg(self):          #Instance Method
        return (self.m1 + self.m2 + self.m3)

    def get_m1(self):       # Accessor Method
        return self.m1

    def set_m1(self):       #Mutator Method
        self.m1 = value

s1 = Student(34,47,32)
s2 = Student(89,32,12)
```

2. Class Method

```
class Student:

    school = "Telusko"

    def __init__(self):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3

    def avg(self):
        return (self.m1 + self.m2 + self.m3)

    @classmethod           #Decorators
    def getSchool(cls):    #Class Method
        return cls.school

s1 = Student (34,47,32)
s2 = Student (89,32,12)
```

3. Static Method:

```
class Student:

    school = "Telusko"

    def __init__(self):
        self.m1 = m1
        self.m2 = m2
```

```

        self.m3 = m3

    def avg(self):
        return (self.m1 + self.m2 + self.m3)

    @staticmethod          #Decorators
    def info:               #Class Method
        print("This is student class...from ABC")

s1 = Student(34,47,32)
s2 = Student(89,32,12)

```

Inner class (Nested class):

```

class Student:

    def __init__(self,name,rollno):
        self.name = name
        self.rollno = rollno
        self.lap = self.Laptop()

    def show(self):
        print(self.name, self.rollno)
        self.lap.show()

    class Laptop():

        def __init__(self):
            self.brand = "HP"
            self.cpu = "i5"
            self.ram = 8

        def show(self):
            print(self.brand, self.cpu, self.ram)

s1 = Student("Navin", 2)
s2 = Student("Jenny",3)

s1.show()
s1.lap.brand
lap1 = s1.lap
lap2 = s2.lap
lap1 = Student.Laptop()

```

Python Inheritance:

- Inheritance allows us to define a class that inherits all the methods and properties from another class.

- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Create a class named **Person**, with **firstname** and **lastname** properties, and a **printname** method:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the
printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Create a class named **Student**, which will inherit the properties and methods from the **Person** class:

```
class Student(Person):
    pass
```

Note: Use the **pass** keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Use the **Student** class to create an object, and then execute the **printname** method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Add the `__init__()` Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the `super()` Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Add a property called `graduationyear` to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
```

In the example below, the year `2019` should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the `__init__()` function:

Add a `year` parameter, and pass the correct year when creating objects:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Add a method called `welcome` to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of",
              self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Overriding Methods from the Parent Class:

You can override any Method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Importing Classes:

Python lets you store classes in Modules and then import the classes you need into your main program.

Importing a single class:

To make a class

```
class Car:

    def __init__(self):
        self.mil = 10          # Instance Variable
        self.com = "BMW"

c1 = Car()
c2 = Car()

print (c1.com, c1.mil)
print (c2.com, c2.mil)
```

Now importing that class into your program and create an instance from that class:

```
from Car import Car

my_new_car = Car()

print (my_new_car.com, my_new_car.mil)
```

Storing Multiple Classes in a Module:

You can store as many classes as you need in a single module, although each class in a module should be related somehow.

Importing Multiple Classes from a Module:

You can import as many classes as you need into a program file.

```
From car import Car, ElectricCar
```

Importing an Entire Module:

```
import car
```

Importing all classes from a module:

```
from module_name import*
```

Using Aliases:

```
From electric_car import ElectricCar as EC
```

Now you can use this alias whenever you want to make to make an electric car.

```
my_tesla = EC ('tesla', 'roadster', 19)
```