In this notebook we will walk through basics of python programaming which are building blocks for our course.

- Python Language Basics
  - Variables
  - Data types/Casting
  - Strings
  - Binary operators and comparisons
- Data Structures and Sequences
  - Tuple
  - List
  - Set
  - Dictionary
- Statements in Python
  - If,Elif,and Else
  - range()
  - Iterative
- Functions
- lambda function
- Exception Handling

PYTHON VARIABLES

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

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

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

Rules for Python variables:

A variable name must start with a letter or the underscore character

A variable name cannot start with a number

A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

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

Assign Value to Multiple Variables

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

In [ ]:
```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

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

In [ ]:
```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

In [ ]:
```

```

Data Types

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

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

Text Type: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

Getting the Data Type

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

In [ ]:
```
x = 5
print(type(x))
```

In [ ]:
```
x = "Hello World" ## str
x = 20            ##int
x = 20.5          ##float
x = 1j            ###complex

x = ["apple", "banana", "cherry"] ### list
x = ("apple", "banana", "cherry") ###tuple

x = range(6)                      ###range

x = {"name" : "John", "age" : 36} ###dict
x = {"apple", "banana", "cherry"} ### set


x = True             ### bool

x = b"Hello"         ### bytes
x = bytearray(5)     ### bytearray

x = memoryview(bytes(5)) ### memoryview
```

Python Casting

In [ ]:
```
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

In [ ]:
```
x = float(1)     # x will be 1.0
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

In [ ]:
```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

NOTE:

In [8]:
```
x=int('3.2')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-63b6d9689243> in <module>
----> 1 x=int('3.2')

ValueError: invalid literal for int() with base 10: '3.2'
```

Strings

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

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

In [ ]:
```
print("Hello")
print('Hello')
```

Multiline Strings

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

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

Strings are Arrays

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

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

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

Slicing

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

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

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

In [ ]:
```

```

Negative Indexing

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

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

String Length

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

It will count the number of elements in agiven string

In [ ]:
```
a = "Hello, World!"
print(len(a))
```

In [ ]:
```

```

Check String

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

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

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

String Concatenation

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

In [ ]:
```
a = "Hello"
b = "World"
c = a + b
print(c)
```

In [ ]:
```
x='5'
y="grade"
z=x+y
print(z)
```

NOTE:

In [1]:
```
x=7
y="Grade"
z=x+y
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-7a4d8069483c> in <module>
      1 x=7
      2 y="Grade"
----> 3 z=x+y

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python Operators

Python Operators Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

Arithmetic operators

Assignment operators

Comparison operators

Logical operators

Identity operators

Membership operators

Bitwise operators

Python Arithmetic Operators

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

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

Python Assignment Operators

Assignment operators are used to assign values to variables:

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

Python Comparison Operators

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

Python Logical Operators

Logical operators are used to combine conditional statements:

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

Python Identity Operators

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

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

Python Membership Operators

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

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

In [ ]:

Python Collections (Arrays)

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

List is a collection which is ordered and changeable. Allows duplicate members.

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

Set is a collection which is unordered and unindexed. No duplicate members.

Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

In [ ]:

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

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

In [ ]:

Access Items

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

In [ ]:

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

In [ ]:

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

In [ ]:

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

In [ ]:
```

```

Range of Indexes

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

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

In [ ]:
```

```

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

In [ ]:
```

```

Range of Negative Indexes

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

In [ ]:
```

```

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

In [ ]:
```

```

Check if Item Exists

To determine if a specified item is present in a list use the in keyword:

In [ ]:
```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

List Length

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

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

Change Item Value

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

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

In [ ]:
```

```

Add Items

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

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

In [ ]:
```

```

you can use the extend() method, which purpose is to add elements from one list to another list:

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

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

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

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

In [ ]:
```

```

To get index of any specific item in the list

In [ ]:
```
thislist = ["apple", "banana", "cherry"]
print(thislist.index("apple"))
```

In [ ]:
```

```

Remove Item

There are several methods to remove items from a list:

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

In [ ]:
```

```

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

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

In [ ]:
```

```

The del keyword removes the specified index:

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

In [ ]:
```

```

The clear() method empties the list:

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

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

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

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

The list() Constructor

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

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

In [ ]:
```

```

List Methods

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

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

In [ ]:
```

```

Tuple

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

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

In [ ]:
```

```

Access Tuple Items

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

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

In [ ]:
```

```

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

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

In [ ]:
```

```

Range of Indexes

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

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

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

In [ ]:
```

```

Change Tuple Values

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

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

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

print(x)
```

In [ ]:
```

```

Add Items

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

In [5]:
```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-fe88f2910dbf> in <module>
      1 thistuple = ("apple", "banana", "cherry")
----> 2 thistuple[3] = "orange" # This will raise an error
      3 print(thistuple)

TypeError: 'tuple' object does not support item assignment
```

Create Tuple With One Item

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

In [ ]:
```
thistuple = ("apple",)
print(type(thistuple))

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

In [ ]:

Tuple Length

To determine how many items a tuple has, use the len() method:

In [ ]:
```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

In [ ]:

The tuple() Constructor

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

In [ ]:
```
thistuple = tuple(["apple", "banana", "cherry"])
print(thistuple)
```

In [ ]:

Tuple Methods

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

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

In [ ]:

Python Sets

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

In [ ]:
```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

In [ ]:

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.

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

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

In [ ]:

Add Items

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

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

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

In [ ]:

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

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

print(thisset)
```

In [ ]:

Get the Length of a Set

To determine how many items a set has, use the len() method.

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

In [ ]:
```
```

Remove Item

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

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

NOTE:

- If the item to remove does not exist, remove() will raise an error.
- If the item to remove does not exist, discard() will NOT raise an error.

You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

SET MEthods

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

In [ ]:

Python Dictionaries

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

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

In [ ]:

Accessing Items

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

In [ ]:
```
x = thisdict["model"]
print(x)
```

In [ ]:

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

In [ ]:
```
x = thisdict.get("model")
print(x)
```

In [ ]:

Change Values

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

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

In [ ]:

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

In [ ]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

In [ ]:

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the len() function.

In [ ]:
```
print(len(thisdict))
```

In [ ]:
```

```

Adding Items

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

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

In [ ]:
```

```

Removing Items

There are several methods to remove items from a dictionary:

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

In [ ]:
```

```

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

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

In [ ]:
```

```

- The clear() method empties the dictionary:

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

In [ ]:
```

```

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy()

In [6]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

In [7]:
```
mydict is thisdict
```

Out[7]:
```
False
```

- Another way to make a copy is to use the built-in function dict().

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

In [ ]:
```

```

Python Dictionaries Methods

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

In [ ]:
```

```

Python If ... Else

Python Conditions and If statements

In [ ]:
```

```

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.

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

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

Indentation

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

In [8]:
```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an erro
```

```
  File "<ipython-input-8-b08ffcd75138>", line 4
    print("b is greater than a") # you will get an erro
        ^
IndentationError: expected an indented block
```

In [ ]:
```

```

Elif

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

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

In [ ]:
```

```

Else

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

In [ ]:
```
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")
```

In [ ]:
```

```

Short Hand If

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

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

In [ ]:
```

```

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:

In [ ]:
```
a = 2
b = 330
print("A") if a > b else print("B")
```

In [ ]:
```

```

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

Multiple else statements on the same line:

In [ ]:
```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

In [ ]:
```

```

And

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

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

In [ ]:
```

```

Or

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

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

In [ ]:
```

```

Nested If

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

In [ ]:
```
x = 41

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

The pass Statement

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

In [ ]:
```
a = 33
b = 200

if b > a:
    pass
```

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

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

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

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

In [ ]:
```

```

The break Statement

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

In [ ]:
```
## Exit the loop when i is 3:
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

The continue Statement

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

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

In [ ]:
```

```

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.

Looping Through a String

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

In [ ]:
```
for x in "banana":
    print(x)
```

In [ ]:
```

```

Loop Through a List

You can loop through the list items by using a for loop:

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

In [ ]:
```

```

Loop Through a Tuple

You can loop through the tuple items by using a for loop.

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

In [ ]:
```

```

Loop through the Set

In [ ]:
```
thisset = {"apple", "banana", "cherry"}

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

In [ ]:
```

```

Loop Through a Dictionary

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

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

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

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

In [ ]:
```

```

- #### Print all values in the dictionary, one by one:

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

The break Statement

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

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

In [ ]:
```

```

The continue Statement

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

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

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":

In [ ]:
```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

In [ ]:
```

```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

In [ ]:
```
range(6)
```

In [ ]:
```
list(range(6))
```

In [ ]:
```
for x in range(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:

In [ ]:
```
range(2, 30, 3)
```

In [ ]:
```

```

Python Functions

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

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

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

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

In [ ]:
```

```

Calling a Function

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

In [ ]:
```
my_function()
```

Arguments

Information can be passed into functions as arguments.

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

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

In [10]:
```
def my_function(x):
    print(x + " is my name")
```

In [ ]:
```
my_function("Jhon")
```

NOTE: If passed more or less argumnets then expected then here function will throw an error

In [11]:
```
my_function()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-7bebf01be998> in <module>
----> 1 my_function()

TypeError: my_function() missing 1 required positional argument: 'x'
```

Parameters or Arguments?

From a function's perspective:

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

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

Arbitrary Arguments, *args

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:

- If the number of arguments is unknown, add a * before the parameter name:

In [ ]:
```
def my_function(*emp):
    print("The newest employee is " + emp[2])

my_function("Emily", "Tom", "Liv")
```

In [ ]:
```

```

Arbitrary Keyword Arguments, **kwargs

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

This way the order of the arguments does not matter.

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.

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

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

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

In [ ]:
```

```

In [ ]:
```
def my_function(**emp):
     print("His last name is " + emp["lname"])

my_function(fname = "Tom", lname = "Jeferson")
```

In [ ]:
```

```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

In [ ]:
```
def my_function(city = "California"):
     print("I am new here I am from  " + city)

my_function("Mumbai")
my_function("Montreal")
my_function()
my_function("Florida")
```

In [ ]:
```

```

Return Values

To let a function return a value, use the return statement:

In [ ]:
```
def my_function(x):
     return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

In [ ]:
```

```

The pass Statement

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

In [ ]:
```
def myfunction():
    pass
```

In [ ]:
```

```

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:

In [ ]:
```
####Add 10 to argument a, and return the result:

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

In [ ]:
```

```

In [ ]:
```
### Multiply argument a with argument b and return the result:

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

In [ ]:
```

```

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:

In [ ]:
```
def myfunc(n):
     return lambda a : a * n
```

In [ ]:
```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
```

In [ ]:
```

```

In [ ]:
```
print(mydoubler(11))
```

In [ ]:
```
```

Lambda with map() and filter()

Q. Double all the elements of given sequence

In [21]:
```
seq=[1,2,3,4,5]
```

METHOD:1

In [15]:
```
seq2=[]

for item in seq:
    seq2.append(item*2)
```

In [16]:
```
print(seq2)
```

```
[2, 4, 6, 8, 10]
```

METHOD:2

In [18]:
```
x=range(len(seq))
print(x)
```

```
range(0, 5)
```

In [22]:
```
for i in x:
    seq[i]*=2
```

In [23]:
```
print(seq)
```

```
[2, 4, 6, 8, 10]
```

METHOD: 3

- List comprehenssion

In [24]:
```
[item*2 for item in seq]
```

Out[24]:
```
[4, 8, 12, 16, 20]
```

METHOD:4

- using lambda function

```
In [25]: map(lambda x:x*2,seq)
Out[25]: <map at 0x188de932fc8>
```

```
In [26]: list(map(lambda x:x*2,seq))
Out[26]: [4, 8, 12, 16, 20]
```

Use of filter()

```
In [ ]: seq=[4,8,9,2,3,10,1]
```

Q. Filter out all those values which are greater than 4

```
In [35]: list(filter(lambda x:x>4,seq))
Out[35]: [6, 8, 10]
```

```
In [ ]:
```

Python Try Except

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

EXCEPTION HANDLING

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement

In [ ]:
```
try:
    print(x)
except:
    print("An exception occurred")
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

In [ ]:
```
try:
    print(k)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

In [ ]:
```

```

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

In [ ]:
```
try:
    print(x)
except:
    print("Something went wrong")
finally:
    print("The 'try except' is finished")
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

In [ ]:
```
Raise an error and stop the program if x is lower than 0:

x = -1

if x < 0:
    raise Exception("Sorry, no numbers below zero")
```

he raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

In [ ]:
```
x = "hello"

if not type(x) is int:
    raise TypeError("Only integers are allowed")
```

In [ ]:
```

```