



Python

Imports

- A library is basically someone else's code.
- Contributors can publish Python libraries to [PyPi](#) (the python package index).
- These libraries can be downloaded using `pip install [package]` or `conda install [package]`, after which they are stored in `PATH`.

```
from utils import m_and_m
m_and_m.mom_spaghetti_2()

# Call a single function
from utils.m_and_m import mom_spaghetti
mom_spaghetti()

# Call everything
from utils import *
```

Important! After importing a module, if the import statement is rerun with the same module, Python will ignore that call because it assumes that it has already run the import statement and will simply load what it ran. Thus, if any change is made to the modules, the kernel must be reset to effect the changes.

Variables

```
x = 3
x + x
x = 10
x

x + 3 # Addition
x - 3 # Subtraction
x * 3 # Multiply
x / 3 # Divide (returns a float)
x ** 3 # Power (and roots)
x % 3 # Modulo (remainder)
x // 3 # Floor division (rounds down to an integer)
```

Keywords

 If you accidentally reassign a python keyword use Kernel —> Restart

```
# List of keywords
import keyword

for i in keyword.kwlist:
    print(i)
```

Booleans

```
bool1 = True
bool2 = False
type(bool1)
```

```
# If its empty it returns false
string = "Hello"
print(bool(string))
empty = ""
print(bool(empty))
```

Booleans function in Python as they do in maths. Comparison operators that return booleans based on whether the comparison is true or not:

- (<) less than
- (>) more than
- (<=) less than or equal to
- (>=) more than or equal to
- (==) equal to (single = is assignment)
- (!=) not equal,

are used to evaluate equality/inequality.

Keywords are employed to chain/modify boolean operators:

- and
- or
- not

```
1 < 2
1 >= 2
1 < 2 and 10 < 20
25 > 37 or 55 < 100
not 100 > 1

# use the 'in' keyword to determine if an item is iterable (also works for strings).
print("x" in [1,2,3])
print("x" in ['x','y','z'])
print("a" in "a world")
```

Strings

<https://docs.python.org/2/library/stdtypes.html#string-methods>

- The backslash (\) character is utilised to escape apostrophes or other special characters in strings.
- Strings are iterable, meaning that they can return their elements one at a time
- Strings are also immutable, meaning that their elements cannot be changed once assigned. Must be **REASSIGNED** to change them
- Each character in a string is one element; this includes spaces and punctuation marks
- Indexing enables us to call back one element
- Slicing enables us to call back a range of elements

```
string = 'Hello world' # Both work
string = "Hello world" # But this is preferable (apostrophes can prematurely end)

print('What's the problem here?') # Like this
print("What\'s the \"problem\" here?") # This also works

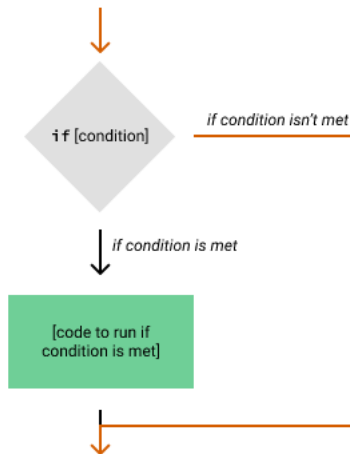
string[0] # 1st character
string[1:4] # Slice between 1 and 4
string[1:] # Slice from 1 onwards
string[:4] # Slice 3 and before
```

```
# This isnt a thing and will not work
my_first_string[0] = 'l'
```

Control Flow

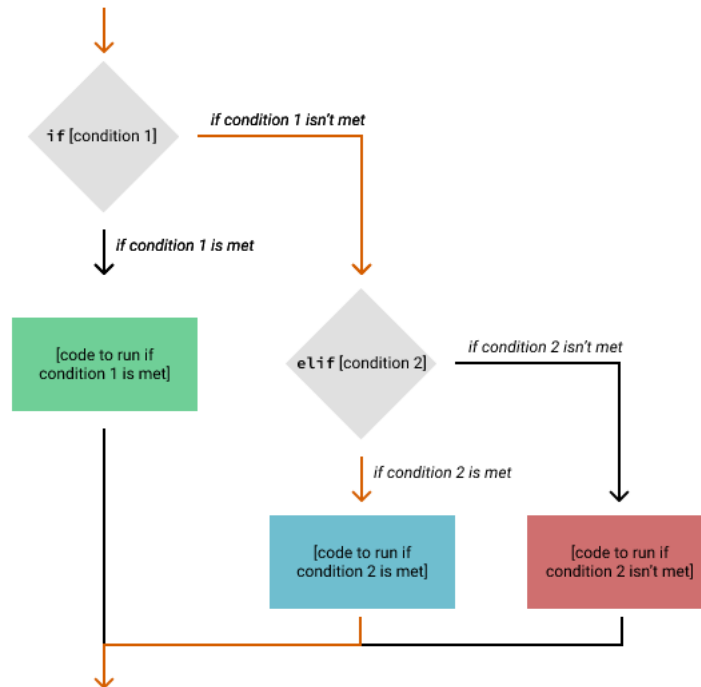
Conditional Statements

If Statements



- An if statement is a piece of code that causes another piece of code to be executed based on the fulfillment of a condition.
- If statements employ boolean operators to determine if a condition is true and whether or not to execute the dependent piece of code subsequently.
- Note the indentation. Python determines precedence based on indentation/whitespace rather than brackets, as in other languages.
- The standard practice for indentation, as recommended by the PEP8 guidelines, is to use four spaces; however, you can also use a tab.
- Spaces and tabs should not be mixed for indentation.
- Jupyter Notebook, similar to many other IDEs, automatically creates an indentation after a colon

Elif And Else Statements



- Elif ('else if') statements add a block of code to be executed if the first condition is not fulfilled, i.e. if the first 'if' statement does not run.
- Else statements add a block of code to be executed if none of the previous conditions are fulfilled, i.e. if the 'if' and 'elif' statements do not run.

```

x = input('Enter your age')
x = int(x) # Input will return a string, you need an integer
if x >= 21:
    print("You are allowed to drink in the US")
elif x >= 18:
    print("You are allowed to drink, but not in the US")
elif x < 0:
    print('Wait, what??')
else:
    print('You can have a Fanta')

```

```

a = int(input('Enter a number for A'))
b = int(input('Enter a number for B'))

print(f'A is equal to {a}')
print(f'B is equal to {b}')
if a > b:
    print("A is larger than B")
elif a < b:
    print("A is smaller than B")
else:
    print("A is equal to B")

```

```

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

```

```

a = int(input('Enter a number for A'))
b = int(input('Enter a number for B'))
c = int(input('Enter a number for C'))

```

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

Lists

Tutorial: <https://www.geeksforgeeks.org/python-get-unique-values-list/>

They serve as ordered collections that enable you to store and organize multiple items in a structured manner.

Ordered: Lists maintain the order of elements as they are added. The position of an element in a list is determined by its index.

Mutable: Lists are mutable, meaning you can modify, add, or remove elements after the list is created. This allows for dynamic changes to the list's content.

Heterogeneous Elements: Lists can contain elements of different data types. You can have a mix of integers, strings, or even other lists within a single list.

Variable Length: Lists can grow or shrink dynamically as elements are added or removed. They do not have a fixed size, allowing for flexibility in managing varying amounts of data. We will see how to do that later in this lesson.

```
# INITIALISATION
[1, 2, 3, 4, 5] # List of ints
["apple", "banana", "cherry"] # List of strings
[10, "hello", 3.14] # List of mixed data types

# Create a list
empty_list = []

# INDEXING

len(list_name) - 1 # Access last item

fruits = ["apple", "banana", "cherry"]
print(fruits[0])    # Access 1st item
print(fruits[1])    # Access 2nd item
print(fruits[-1])   # Access last item
print(fruits[-2])   # Access 2nd last item
print(fruits[3])    # IndexError: list index out of range

index = fruits.index("banana")
print(index)

list_4 = [list_1[3], list_2[3]]
print(list_4)

----Output----
apple
banana
cherry
banana

IndexError: list index out of range
1

[True, 0]
```

```
# SLICING

numbers = [1, 2, 3, 4, 5]
sliced_list = numbers[1:4]
print(sliced_list)

print(numbers[:3])

print(numbers[2:])
print(numbers[:])

----Output----
[2, 3, 4]
```

Data Storage and Organization: Lists are like special containers that can hold lots of things together. They help you store and organize different pieces of data in one place. It's like having a box where you can keep your favorite snacks. Lists make it easier to keep track of and work with collections of data.

Iterating and Manipulating Data: Lists allow you to change, add, or take away things from the list.

Lists are created by enclosing elements within square brackets (`[]`), and each element is separated by a comma.

An index refers to the position or location of an element within the list. Each element in a list is assigned a unique index value that allows you to access and manipulate that element.

Indexing starts from `0`

Negative indexing: where the index `-1` refers to the last element. Negative indices count from the end of the list towards the beginning.

`index()` allows you to find the index of a specific element within a list. It returns the index of the first occurrence of the element in the list.

An `IndexError` occurs when you try to access an index that is outside the valid range of indices for a list. This typically happens when the index is negative or greater than or equal to the length of the list.

```
[1, 2, 3]
[3, 4, 5]
[1, 2, 3, 4, 5]
```

```
# NESTED LISTS
```

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list[0])
print(nested_list[1][1])
```

```
nested_list[1][0] = 10
print(nested_list)
```

```
list_3 = [list_1, list_2]
print(list_3)
```

```
----Output----
```

```
[1, 2, 3]
5
```

```
[[1, 2, 3], [10, 5, 6], [7, 8, 9]]
```

```
[[1, 1.1, 'one', True], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
# LIST OPERATIONS
```

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
fruits = ["apple", "banana", "cherry"]
```

```
# Concatenation
concatenated = list1 + list2
print(concatenated)
```

```
# Repetition
repeated = list1 * 3
print(repeated)
```

```
list_2 = [0]*10
print(list_2)
```

```
# Membership Testing
print("banana" in fruits)
print("orange" not in fruits)
```

```
----Output----
```

```
[1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
True
True
```

```
# LIST MANIPULATION
```

```
numbers = [10, 20, 30, 40, 50]
fruits = ["apple", "banana", "cherry"]
```

```
# Modifying
numbers[2] = 35
print(numbers)
```

```
# Adding Elements
fruits.append("orange")
print(fruits)
fruits.insert(1, "orange")
print(fruits)
more_fruits = ["orange", "mango"]
fruits.extend(more_fruits)
print(fruits)
```

Slicing allows you to extract a portion of a list by specifying a range of indices. It creates a new list containing the selected elements. The syntax for slicing is `start_index: stop_index`.

The `start_index` parameter denotes the index at which the slicing operation begins. The `stop_index` parameter denotes the index at which the slicing operation ends.

The element at the `start_index` is included in the slice. The element at the `stop_index` is not included in the slice.

Omitting indices: When you omit the `start_index` in a slice, it defaults to `0`, and when you omit the `stop_index`, it defaults to the length of the list.

Nesting Lists allows you to create complex data structures that can store multiple levels of information. Each nested list can have its own elements, including other nested lists. This hierarchical structure enables you to represent data in a more organized and structured manner.

To create a nested list include lists as elements within another list

`nested_list` contains three inner lists, each representing a row of values. The outer list encapsulates these inner lists, forming a nested structure.

Access elements in a nested list by using multiple index positions. The first index represents the outer list's position, and the second index represents the position within the inner list

Concatenated using `+`. It combines two or more lists into a single list, preserving the order. `+`

concatenates `list1` and `list2`, resulting in a new list

```
concatenated
```

Lists can be repeated or multiplied using the `*`. It creates a new list by repeating the elements of the original list a specified number of times. `*` repeats the elements of `list1` list 3 times, creating a new list named `repeated`.

You can test whether an element is present in a list using

`in` and `not in`. They return a Boolean (`True` or `False`) based on the presence or absence. `in` checks if `"banana"` is present

```
# Removing Elements
fruits.remove("banana")
print(fruits)
fruits.remove("grape") # ValueError

removed_number = numbers.pop(2)
print(removed_number)
print(numbers)

----Output----
[10, 20, 35, 40, 50]

["apple", "banana", "cherry", "orange"]
["apple", "orange", "banana", "cherry"]
["apple", "banana", "cherry", "orange", "mango"]

["apple", "cherry"]
ValueError: list.remove(x): x not in list

30
[10, 20, 40, 50]
```

```
# SORTING A LIST

numbers = [5, 2, 8, 1, 9]
fruits = ["apple", "banana", "cherry"]

numbers.sort()
print(numbers)
fruits.sort(reverse=True)
print(fruits)

----Output----
[1, 2, 5, 8, 9]
["cherry", "banana", "apple"]
```

```
# REVERSING A LIST

numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(numbers)

----Output----
[5, 4, 3, 2, 1]
```

```
# MIN + MAX
numbers = [5, 2, 8, 1, 9]

print(min(numbers))
print(max(numbers))

----Output----
1
9
```

```
# LENGTH OF A LIST

fruits = ["apple", "banana", "cherry"]

length = len(fruits)
print(length)

----Output----
3
```

```
# JOINING LISTS

fruits = ["apple", "banana", "cherry"]
```

in the `fruits` list, returning `True`. `not in` checks if `"orange"` is not present in the `fruits` list, returning `True`.

Lists support various operations to interact with elements. Lists are mutable, you can modify the value by assigning a new value to its index.

`append()` method is used to add an element to the end of a list. It takes a single argument, which is the value to be added.

`insert()` method allows you to add an element at a specific index within a list. It takes two arguments: the index at which to insert the element and the value to be inserted. `insert()` adds the value `"orange"` at index `1` in `fruits`. The existing elements are shifted to accommodate the new element.

`extend()` allows you to add multiple elements from another list to the end of the current list. It takes an iterable as an argument, like another list or a string. `extend()` is used to add the elements from the `more_fruits` list to the `fruits` list. The resulting list contains all the elements from both lists.

`remove()` removes the first occurrence of a specified value from the list. It searches for the value and removes it if found. If the value is not present in the list, a `ValueError` is raised. `fruits` contains 3 elements: `"apple"`, `"banana"`, and `"cherry"`. Trying to remove `"grape"` using `remove()` causes a `ValueError` because `"grape"` is not in the list. `remove()` expects the value to exist in the list and removes only the first occurrence of that value.

`pop()` removes an element at a given index and returns its value. If no index is specified, it removes and returns the last element allowing you to remove elements and capture their values for further processing. If you try to `pop()` an index that is out of range, an `IndexError` will be raised.

```
joined_string = ", ".join(fruits)
print(joined_string)
```

```
----Output----
"apple, banana, cherry"
```

`sort()` allows you to sort the elements in ascending order by default. You can also specify the `reverse=True` argument to sort the list in descending order. `sort()` is used to sort the `numbers` list in ascending order and the `fruits` list in descending order.

Reversing a list involves changing the order of its elements. `reverse()` allows you to reverse the order of elements in-place.

To find the minimum and maximum use the `min()` and `max()` functions. These functions return the smallest and largest elements in the list.

`len()` function in Python allows you to determine the length or number of elements in a list. It returns an integer representing the count of elements in the list.

`join()` allows you to concatenate the elements of a list into a single string. It takes a string as an argument and joins the elements of the list using that string as a separator.

```
# FIND UNIQUE VALUES IN A LIST

my_list = [1, 2, 3, 3, 3, 4, 5]

def unique(original_list):
    unique_list = []

    for x in original_list:
        if x not in unique_list:
            unique_list.append(x)

    if unique_list == 0:
        print("All elements are unique")
    else:
        print("There are duplicate elements")
```

```
# PLAYING WITH LISTS

import random
import string

usernames = []

for i in range(10):
    length = random.randint(5, 10)
    username = ''.join(random.choices(string.ascii_lowercase + string.digits, k=length))
    usernames.append(username)

print(usernames)

# Print the datatype of usernames
print(type(usernames))

# Print the length of usernames
print(len(usernames))

# Print the datatype of the first element of usernames
print(type(usernames[0]))

# Make a new list called usernames_2 containing the last 5 elements of
usernames_2 = usernames[-5:]

# Remove the second element of usernames_2 and assign it to a variable called user_example
user_example = usernames_2.pop(1)
print(user_example)

# Using sort(), sort the elements of usernames alphabetically
usernames.sort()
print(usernames)

# Find the index of user_example in usernames and assign it to idx_user_example
idx_user_example = usernames.index(user_example)
# Retrieve the corresponding element in usernames, convert it to uppercase and assign it to upper_user_example
upper_user_example = usernames[idx_user_example].upper()
```

```
# Replace the corresponding element in usernames with upper_user_example
usernames[idx_user_example] = upper_user_example
```

```
number_plates = ["G06 WTR", "WL11 WFL", "QW68 PQR"]

year = number_plates[0][1:3]
year = int(year)
```

```
my_list_1 = [1, 2, 3, 4]
my_list_2 = [5, 6, 7, 8]

my_list_1.append(my_list_2)
print(my_list_1)
print(my_list_2)

my_list_1.extend(my_list_2)
print(my_list_1)
print(my_list_2)

----Output-----
[1, 2, 3, 4, [5, 6, 7, 8]]
[5, 6, 7, 8]

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

Dictionaries

Unlike lists that use numbers to keep track of things, dictionaries allow you to use any special word or number as a key. It's like having a personalized filing system where you can organize your things in different ways

A dictionary is an ordered collection of *key-value pairs*, where each key is unique within the dictionary. Each key in a dictionary maps to a corresponding value. Key-value pairs are essential for organizing and accessing data efficiently in dictionaries.

- Keys must be unique within a dictionary. Duplicate keys are not allowed. If you attempt to add a key-value pair with a key that already exists, the existing value will be overwritten.
- Keys must be immutable objects. This means that once a key is assigned, it cannot be changed. Common examples of valid keys are strings and numbers.
- Mutable objects like lists or dictionaries cannot be used as keys since they can change their value, which would make it difficult to maintain uniqueness and retrieve values efficiently
- Values in dictionaries have no restrictions. They can be of any type and can include numbers, strings, lists, other dictionaries, or custom objects.

```
# CREATE A DICTIONARY

empty_dict = {}
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}
student_scores = dict(Alice=90, Bob=85, Charlie=95)
```

An empty dictionary is created with no initial key-value pairs. The `student_scores` dictionary contains three key-value pairs. In this dictionary `"Alice"`, `"Bob"`, and `"Charlie"` are keys, and `90`, `85`, and `95` are their corresponding values.

```
# ACCESSING ELEMENTS IN A DICTIONARY
```

```
# NESTED DICTIONARIES

student_info = {
    "Alice": {"age": 20, "grade": "A"},
    "Bob": {"age": 19, "grade": "B"},
    "Charlie": {"age": 21, "grade": "A+"}
}

alice_age = student_info["Alice"]["age"]
bob_grade = student_info["Bob"]["grade"]

print(alice_age)
print(bob_grade)

-----Output-----
20
B
```

```
# ADDING NEW KEYS TO A DICTIONARY

student_scores = {"Alice": 90, "Bob": 85}

# Adding a new key-value pair
student_scores["Charlie"] = 95

print(student_scores)

-----Output-----
{'Alice': 90, 'Bob': 85, 'Charlie': 95}
```

The `pop()` method removes a key-value pair from a dictionary and returns the value associated with the specified key

The `popitem()` method removes and returns an arbitrary key-value pair from the dictionary. Unlike `pop()`, `popitem()` doesn't require you to provide a key; it automatically removes the last inserted key-value pair.

```
alice_score = student_scores["Alice"]
print(alice_score)

-----Output-----
90
```

Dictionaries can also contain nested dictionaries as their values. This allows you to create hierarchical data structures. To access values within nested dictionaries, you can chain multiple square bracket notations.

```
# MODIFY A DICTIONARY

student_scores["Charlie"] = 98
print(student_scores["Charlie"])

-----Output-----
98
```

Dictionaries in Python provide straightforward ways to add new keys or modify existing keys along with their corresponding values. To add a new key-value pair to a dictionary, you can assign a value to a previously non-existent key. If the key already exists, the assigned value will overwrite the existing value.

```
# REMOVING ELEMENTS FROM A DICTIONARY

student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}
bob_score = student_scores.pop("Bob")
print(student_scores)
print(bob_score)

removed_item = student_scores.popitem()
print(student_scores)
print(removed_item)

-----Output-----
{'Alice': 90, 'Charlie': 95}
```

```
# CONVERTING DICTIONARIES TO A LIST

student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

# Converting to a list of keys
keys_list = list(student_scores.keys())
print(keys_list)

# Converting to a list of values
values_list = list(student_scores.values())
print(values_list)

# Converting to a list of key-value pairs
items_list = list(student_scores.items())
print(items_list)

student_scores = dict(items_list)
print(student_scores)

-----Output-----
['Alice', 'Bob', 'Charlie']

[90, 85, 95]

[('Alice', 90), ('Bob', 85), ('Charlie', 95)]

{'Alice': 90, 'Bob': 85, 'Charlie': 95}
```

85

```
{'Alice': 90, 'Bob': 85}
{'Charlie': 95}
```

You can convert a dictionary into a list of its keys, values, or key-value pairs using the `keys()`, `values()`, and `items()` methods.

Conversely, you can convert a list of key-value pairs into a dictionary using the `dict()` constructor.

The `in` operator provides a convenient way to determine whether a specific key exists in a dictionary or not. The `in` operator returns a boolean value (`True` or `False`) based on the presence or absence of the key.

- `print("Alice" in student_scores)` returns `True` because the key `"Alice"` exists in the `student_scores` dictionary
- `print("John" in student_scores)` returns `False` because the key `"John"` does not exist in the `student_scores` dictionary

```
# FINDING AN ELEMENT IN A DICTIONARY

student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

# Checking for the presence of keys
print("Alice" in student_scores)
print("John" in student_scores)

-----Output-----
True
False
```

copy()

The `copy()` method creates a copy of the dictionary. It returns a new dictionary with the same key-value pairs as the original dictionary.

```
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

copied_dict = student_scores.copy()

print(copied_dict)

-----Output-----
{'Alice': 90, 'Bob': 85, 'Charlie': 95}
```

get()

The `get()` method retrieves the value associated with the given key. If the key is not found, it returns the specified default value (or `None` if not provided).

```
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

alice_score = student_scores.get("Alice")
john_score = student_scores.get("John", 0)

print(alice_score)
print(john_score)

----Output-----
90
0
```

items()

The `items()` method returns a view object that contains tuples of key-value pairs present in the dictionary.

```
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

items_view = student_scores.items()

print(items_view)

----Output-----
dict_items([('Alice', 90), ('Bob', 85), ('Charlie', 95)])
```

keys()

The `keys()` method returns a view object that contains all the keys present in the dictionary.

```
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

keys_view = student_scores.keys()

print(keys_view)

----Output-----
dict_keys(['Alice', 'Bob', 'Charlie'])
```

update()

The `update()` method merges another dictionary into the existing dictionary. If there are common keys, the values from the second dictionary overwrite the values in the first dictionary.

```
dict1 = {"A": 1, "B": 2}
dict2 = {"C": 3, "D": 4}

dict1.update(dict2)

print(dict1)

dict1 = {"A": 1, "B": 2}
dict3 = {"B": 3, "C": 4}

dict1.update(dict3)

print(dict1)

----Output-----
{'A': 1, 'B': 2, 'C': 3, 'D': 4}

{'A': 1, 'B': 3, 'C': 4}
```

values()

The `values()` method returns a view object that contains all the values present in the dictionary.

```
student_scores = {"Alice": 90, "Bob": 85, "Charlie": 95}

values_view = student_scores.values()

print(values_view)

---- -Output-----
dict_values([90, 85, 95])
```

Functions

append()

The `append()` method is used to add an element to the end of a list. It takes a single argument, which is the value to be added.

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

---- -Output-----
["apple", "banana", "cherry", "orange"]
```

extend()

The `extend()` method allows you to add multiple elements from another list to the end of the current list. It takes an iterable as an argument, such as another list or a string.

```
fruits = ["apple", "banana", "cherry"]
more_fruits = ["orange", "mango"]
fruits.extend(more_fruits)
print(fruits)

---- -Output-----
["apple", "banana", "cherry", "orange", "mango"]
```

index()

The `index()` allows you to find the index of a specific element within a list. It returns the index of the first occurrence of the element in the list.

```
fruits = ["apple", "banana", "cherry"]

index = fruits.index("banana")
print(index)

---- -Output-----
1
```

insert()

The `insert()` method allows you to add an element at a specific index within a list. It takes two arguments: the index at which to insert the element and the value to be inserted.

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

----Output-----
["apple", "orange", "banana", "cherry"]
```

join()

`join()` allows you to concatenate the elements of a list into a single string. It takes a string as an argument and joins the elements of the list using that string as a separator.

```
fruits = ["apple", "banana", "cherry"]

joined_string = ", ".join(fruits)
print(joined_string)

----Output-----
"apple, banana, cherry"
```

len()

`len()` function in Python allows you to determine the length or number of elements in a list. It returns an integer representing the count of elements in the list.

```
# LENGTH OF A STRING
word = "Hello World"

len("Hello World")
len(word)

# LENGTH OF A LIST
fruits = ["apple", "banana", "cherry"]

length = len(fruits)
print(length)

----Output-----
11
11
3
```

min() max()

To find the minimum and maximum use the `min()` and `max()` functions. These functions return the smallest and largest elements in the list.

```
# MIN + MAX
numbers = [5, 2, 8, 1, 9]

print(min(numbers))
print(max(numbers))

----Output-----
```

```
1
9
```

remove()

`remove()` removes the first occurrence of a specified value from the list. It searches for the value and removes it if found. If the value is not present in the list, a `ValueError` is raised.

```
fruits = ["apple", "banana", "cherry"]

fruits.remove("banana")
print(fruits)

fruits.remove("grape") # ValueError: list.remove(x): x not in list

---- -Output-----
["apple", "cherry"]
```

replace()

```
# Replace x with y
replace("x", "y")
```

reverse()

Reversing a list involves changing the order of its elements. `reverse()` allows you to reverse the order of elements in-place.

```
numbers = [1, 2, 3, 4, 5]

numbers.reverse()
print(numbers)

---- -Output-----
[5, 4, 3, 2, 1]
```

sort()

`sort()` allows you to sort the elements in ascending order by default. You can also specify the `reverse=True` argument to sort the list in descending order.

```
numbers = [5, 2, 8, 1, 9]
numbers.sort()
print(numbers)

fruits = ["apple", "banana", "cherry"]
fruits.sort(reverse=True)
print(fruits)

---- -Output-----
[1, 2, 5, 8, 9]
["cherry", "banana", "apple"]
```

pop()

`pop()` removes an element at a given index and returns its value. If no index is specified, it removes and returns the last element allowing you to remove elements and capture their values for further processing. If you try to `pop()` an index that is out of range, an `IndexError` will be raised.

If you try to `pop()` an index that is out of range, an `IndexError` will be raised.

```
numbers = [10, 20, 30, 40, 50]

removed_number = numbers.pop(2)
print(removed_number)
print(numbers)

----Output----
30
[10, 20, 40, 50]
```

print()

Interprets the escape characters (tabs, new lines, etc.) and displays the string without quotations.

```
# Displays the contents of ()
print("Hello World")
print ("Hello" + "World")
print(f"Hello {name}")
```

Round()

```
# Round to 2 decimal places
round(answer, 2)
```

Type()

```
# Show the type of variable
x = 10
type(x)
```

String Functions

Capitalize()

```
# Turns the 1st character in the string to UPPERCASE
z = "how are you?"
z.capitalize()
```

Format()

```
# Inserts data into a string (default in order)
print("The {} {} {}".format("fox", "brown", "quick"))
print("The {2} {1} {0}".format("fox", "brown", "quick"))
print("The {q} {b} {f}".format(f="fox", b="brown", q="quick")) # More readable

# create long decimal
result = 100/777
print(result, end = "\n\n")

# use value:width.precisionf for formatting
```

```
# width is minimum length of string, padded with whitespace if necessary
# precision is decimal places
print("The result was {:.3f}".format(result))
print("The result was {r:1.3f}".format(r=result))
print("The result was {r:1.7f}".format(r=result))
print("The result was {r:.3f}".format(r=result))
```

Lower()

create long decimal

```
result = 100/777
print(result, end = "\n\n")
```

use value:width.precisionf for formatting

width is minimum length of string, padded with whitespace if necessary

precision is decimal places

```
print("The result was {:.3f}".format(result))
print("The result was {r:1.3f}".format(r=result))
print("The result was {r:1.7f}".format(r=result))
print("The result was {r:.3f}".format(r=result))
```

```
# Transforms to LOWERCASE
print(x.lower())
x = x.lower()
```

Split()

```
# Splits a string using a specified separator (space as default)
print(x.split())
print(x.split(","))
```

Upper()

```
# Transform to UPPERCASE
print(x.upper())
x = x.upper()
```

Library's

Math

```
# In-built math library
import math

math.pi
math.sqrt(7)
```

BASICS

Comments are descriptions that help programmers better understand the intent and functionality of the program. They are completely ignored by the Python interpreter.

```
# This is how to comment a single line

'''
Use to comment
multiple lines
'''

# If we do not assign strings to any variable, they act as comments
"This is a comment"
variable = "But this is a string"
```

Whenever string literals are present just after the definition of a function, module, class or method, they are associated with the object as their `__doc__` attribute. We can later use this attribute to retrieve this docstring

Standard Convention:

- Even though they are single-lined, we still use the triple quotes around these docstrings as they can be expanded easily later.
- The closing quotes are on the same line as the opening quotes.
- There's no blank line either before or after the docstring.
- They should not be descriptive, rather they must follow "Do this, return that" structure ending with a period.
- For a function or method should summarize its behavior and document its arguments and return values.
- It should also list all the exceptions that can be raised and other optional arguments

You can also use `help()` to access a docstring

```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2

print(square.__doc__) # Print a docstring
print(print.__doc__) # See docstrings for preexisting functions
import pickle
print(pickle.__doc__) # Classes will also list the functions they contain
-----
def multiplier(a, b):
    """Takes in two numbers, returns their product."""
    return a*b
-----
help(square) # The help function also access the docstring
```

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description

Standard Convention:

- The docstrings for classes should summarize its behavior and list the public methods and instance variables.
- The subclasses, constructors, and methods should each have their own docstrings
- The docstrings for classes should summarize its behavior and list the public methods and instance variables.
- The subclasses, constructors, and methods should each have their own docstrings
- The docstrings for Python script should document the script's functions and command-line syntax as a usable message.
- It should serve as a quick reference to all the functions and arguments.

```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers

    Parameters:
        a (int): A decimal integer
        b (int): Another decimal integer

    Returns:
        binary_sum (str): Binary sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum

print(add_binary.__doc__)

class Person:
    """
    A class to represent a person.
    """

    Attributes
    -----
    name : str
        first name of the person
    surname : str
        family name of the person
    age : int
        age of the person

    Methods
    -----
```

```

info(additional=""):
    Prints the person's name and age.
"""

def __init__(self, name, surname, age):
    """
    Constructs all the necessary attrit

    Parameters
    -----
        name : str
            first name of the person
        surname : str
            family name of the person
        age : int
            age of the person
    """

    self.name = name
    self.surname = surname
    self.age = age

print(Person.__doc__)

```

```

print('This is the print function')
print("This is also a print function")
print('Note how to use escape characters in some words like we\'re to escape the default function')
print('Concatanation '+'also works')
print('Concatanation also works','like this.' , 'The space is automatic.')
print('This will print out "Hi 5",Hi, 5')
print('So will this. Hi '+str(5))
print(int('8')+5, 'This now adds them together')
-----
print(exampleVariable)
print('Number 1 is', number1)
-----
# Concatenate string and float
cone_volume = 7.2
print("The volume of cone is " + str(cone_volume))

```

```

1+3
3-1
1*3
1/3
4**4 '''4 to the power of 4'''
-----
# Percentages 20% VAT
vat = 20
shopping = 10.50
total_price = shopping / vat
shopping_plus_vat = round(shopping + total_price, 2)
-----
# Volume and surface area
def cone(h, r):
    cone_volume = math.pi * r**2 * h / 3
    cone_surface_area = (r + math.sqrt(h**2 + r**2)) * (math.pi * r)
    print("The volume of cuboid is " + str(cone_volume) + " And the surface areas is " + str(cone_surface_area))

```

```

exampleVariable = 55
example-variable = 55+32
EXAMPLE_VARIABLE = print ('Variables can contain functions')
x, y = (3, 5)
x, y, z = (3, 5) '''It will try to assign and equal number to each variable so this will error'''
condition += 1 ''' += adds 1 to the variable. The same as condition = condition + 1'''
-----
x = 6 '''Not a global function, it's just 'comitted to memory' from the begining'''
def example():
    print(x) '''works'''
    print(x+5) '''works'''
    x+=6 '''brakes'''
example()

```

```

'''try this'''
x = 6
def example():
    global x
    print(x)
    x+=5
    print(x)
example()

'''or this'''
x = 6
def example():
    globx = x '''local variable'''
    print(globx)
    globx+=5
    print(globx)
    return globx
x = example()
print(x)
-----
import statistics

example_list = [1,2,3,4,5,6,7,8,9,10]

x = statistics.mean(example_list) '''mean, mode, standard deviation, variance'''
print(x)
-----
my_sum = 5 % 3 # % what is the remainder? 2
#x//y # divided by and rounded down to the nearest integer
2 ** 3 # 2 to the power of 3
2 ** 0.5 # root
round(10/3, 4) # 10 / 3 rounding down to 4 decimals 3.3333
s = 'string' * 3
print(s) # prints string 3 times

```

```

x = 5
y = 10
z = 5
a = 3

x < y '''x is smaller than y'''
z < y > x '''y is bigger than x and z'''
z < y > x > a '''This can get complicated'''
z <= x '''z is smaller or equal to x'''
z = x '''Errors. Does the variable equal the variable or the value equal the value?'''
z == x '''This works'''
z != x '''Not equal'''

```

```

condition = 1

while variable < 10:
    print(condition)
    condition += 1 ''' += adds 1 to the variable. The same as condition = condition + 1'''
    -----
while True:
    print('This is an infinite loop so it will always run. Use CTRL C to brake while running')
    -----
print("Hiya")
    break
for item in items:
    print(items)
for idx in range():
    print(idx)
    if idx == 5: # check if they are the same
        break
    if idx == 4:
        continue

```

```

age = 32
age = 33
print("age")
height = 5.4
name = "Millie"

```

```

staff = False
student = True
items = ("pizza", "plate", "monster", 15.45)
items[0]
parents = ("mom", "dad")
person{
    'name': 'Millie',
    'age': age, #key:value
}
millies_name = person['name']

```

```

exampleList = [1, 2, 3, 4, 5]

for eachNumber in exampleList: ''will allocate each value to eachNumber''
    print(eachNumber)
    print('This line will show every time it prints eachNumber')
print('But this line will show one the loop has finished becaus it isnt indented')
-----
for x in range(1, 11): ''only prints eachNumber between 1 and 11''
    print(x)

```

```

long_word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'
print(long_word)

length = len(long_word) ''get length''
print(length)

first_c = long_word[0] ''get first character''
print(first_c)

last_c = long_word[44] ''get last character''
print(last_c)

add = first_c + last_c ''concatenate letters''
print(add)

```

```

x = 5
y = 10
z = 5

if x < y:
    print('x is smaller than y')

if z < y > x:
    print('y is bigger than x and z')

if z <= x:
    print('z is smaller or equal to x')
-----
if x > y:
    print('x is smaller than y')
else: ''Links to the last if''
    print('x is not smaller than y')

if x > y:
    print('x is smaller than y')
if x > 55:
    print('x is bigger than 55')
else: ''Links to the last if''
    print('x is not smaller than y')
-----
if x > y:
    print('x is bigger than y')
elif x < z:
    print('x is smaller than z')
elif 5 > 2:
    print('5 is bigger than 2')
else:
    print('neither are true')
-----
''As soon as it finds a true statement it will stop looking''
''It will only run the true statement''
-----

```

```

if student:
    print("Hi student")
elif student or age < 18:
    print("not staff")
elif staff:
    print("not student")
elif 'monster' in items:
    print("must be a student")
elif not staff or student:
    print("Hi stranger")
else:
    print("Hi staff")

```

```

'''Each element in a list is an item'''
'''Support indexing and slicing'''
'''Can nest lists within each other'''
'''Mutable: can be changed after creation'''
'''Ordered: has a fixed order and so can be indexed using numbers'''
-----
letters = ["a", "b", "c"]
my_list = [3, "three", 3.0, True]
-----
sentence = 'This is a sentence'
sentence.split('s')
-----
# Check datatype of an element
type(3) # Check datatype of an element
type("three")
type(my_list)
-----
# INDEXING AND SLICING
my_list = ['John', 'Paul', 'George', 'Ringo']
my_list
my_list[0] # First element
my_list[0:3] # Second and third element
my_list[1:] # Second and up
my_list[:3] # First to forth
my_list[::2] # Step size of 2
print(my_list[-1]) # Start from the end
print(my_list[::-1]) # Go backwards

my_list[1] = my_list[1].upper() # Reassign and element
my_list + ['Yoko'] # Adds an element but doesn't change the list
my_list = my_list + ['Yoko'] # Changes the list
my_list = my_list[:4] # Reassign to a slice of the original
print(my_list * 2) # Prints it twice (reassign it if you want it to be permanent)

lst_1 = [1, 2, 3]
lst_2 = [4, 5, 6]
lst_3 = [7, 8, 9]
nest_list = [lst_1, lst_2, lst_3] # A list of lists (nested list)
nest_list[1] # Second list
nest_list[1][1] # Second item of the second list
-----
# FUNCTIONS + METHODS
len(my_list) # Counts the number of items in the list
print(min(num_list)) # Finds the highest or lowest number
print(max(my_list)) # Or the last alphabetically
my_list.sort() # Reorders list alphabetically
num_list.sort() # Or numerically
my_list.reverse() # Reorders in reverse

my_list.append(['Lennon', 'McCartney', 'Harrison', 'Starr']) # Adds items to the end
my_list.extend(['Lennon', 'McCartney', 'Harrison', 'Starr']) # Adds items in a list to the end
my_list.insert(1, 'Lennon') # Adds an item to a specific location
last_item = my_list.pop() # Removes the last item AND returns it
my_list.pop(2) # Removes a specific item
my_list.remove('Yoko') # Finds and Removes an item
str_list = ["This", "is", "a", "sentence."]
print("Joined it".join(str_list)) # Joins all the items together
idx = my_list.index('Lennon') # Returns the index number of the found item

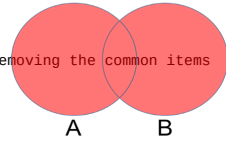
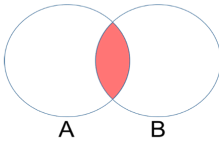
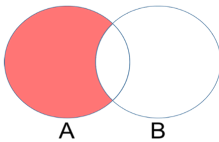
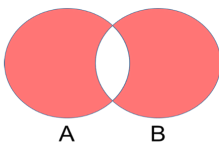
```

```

# SETS (are unordered)
my_set = set([1, 2, 3, 4, 4, 4, 6]) # Will remove the repeated items
my_set = {1, 2, 3, 4, 4, 4, 6} # Does the same thing

```

```
len(my_set) # Returns the size of the set
min(my_set) # Returns the smallest number
my_set.add(1) # Adds something to the set (unless it is repeated)
final_set = set1.union(set2) # Add all elements to another set
final_set = set1.intersection(set2) # Returns a set with all items in c
final_set = set1.difference(set2) # Returns a set with the items in set
final_set = set1.symmetric_difference(set2) # Returns a set with items
```

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$, is the set of all values that are a member of A, or B, or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B.
Difference		$A \setminus B$, is the set of all values of A that are not members of B
Symmetric Difference		$A \triangle B$, is the set of all values which are in one of the sets, but not both.

Classes

```
class calculator:

    def addition(x, y):
        added = x + y
        print(added)

    def subtraction(x, y):
        sub = x - y
        print(sub)

    def multiplication(x, y):
        mult = x * y
        print(mult)

    def division(x, y):
        div = x / y
        print(div)

calculator.multiplication(3, 5)
```

Error Handling

When raising a new exception while another exception is already being handled, the new exception's `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used

This implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

```
try:
    x+y
except TypeError:
    print("Type Error")
finally:
    print("finally blocks are always executed")
```


the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised

FUNCTIONS

```
def exampleFunction():
    '''This is inside the function'''
    '''This is also inside the function'''

    '''This is not'''
    print('This is a function')
    z = 1+2
    print(z)
    '''Nothing will happen because the function wasnt called, just defined'''
    exampleFunction() '''Now it will run'''
    -----
def simpleAddition(number1, number2):
    answer = number1 + number2
    print('Number 1 is', number1)
    print(anmswer)

simpleAddition[1,2]
simpleAddition[number2=2, number1=1] '''These two line are the same'''

def exampleFunction(number1, number2):
def exampleFunction(number1, number2=2): '''These two lines are also the same'''
    print('number1, number 2')

exampleFunction(1)
```

```
for x in range(1, 11): '''only prints eachNumber between 1 and 11'''
    print(x)

'''Works by literally making a list from 1 to 11. Thus very big ranges will effect memory'''
range() '''works by making a generator so can be used for big ranges'''
```

```
def basic_window(width, height, font='TNR',
                 bgc='w', scrollbar=True):
    '''TNR is times new roman'''
    '''bgc is background colour. w is white'''
    basic_window(500, 350 bgc='b') '''width, height, background colour'''
```

input()

Get user input

```
x = input('What is your name? ')

print('Hello', x)
```

Modules

```
pi@raspberrypi ~ $ sudo apt-get install python-matplotlib:
C:\Users\H>C:\Python34\Scripts\pip install matplotlib
```

Writing/Appending/Reading Files

```
text = 'Sample Text to Save\nNew line!'

saveFile = open('exampleFile.txt', 'w') '''if it doesnt exist already, it will create it'''
saveFile.write(text)
saveFile.close()
'''Will be put in the same directry as scripts'''
-----
appendMe = '\nNew bit of information'

appendFile = open('exampleFile.txt', 'a') '''a for append'''
appendFile.write(appendMe)
appendFile.close()
-----
readMe = open('exampleFile.txt', 'r').read() '''r for read. .readlines() creates a list'''

print(readMe)
```

Examples

```
usernames = ['a', 'b', 'c']
print(type(usernames))
print(len(usernames))
print(type(usernames[0]))
print(len(usernames[2]))
```

```
list1 = ['', '', '', '', '']
list2 = ['0'] * 10
list3 = [list1, list2]
list4 = [list3[0][3], list3[1][3]]
print(list4)
```

```
names = ['Millie', 'Sophie']
name1 = set({})
name2 = set({})

for i in names[0]:
    name1.add(i)

for i in names[1]:
    name2.add(i)

common = name1.intersection(name2)
print(common)
```

```
phrase1 = 'Clean Couch'
phrase2 = 'Giant Table'
```

```
phrase1List = []
for i in phrase1:
    phrase1List.append(i)
```

```
phrase2List = []
for i in phrase2:
    phrase2List.append(i)
```

```
if phrase1List[0] == phrase2List[0]:
    print("Same First Letter")
else:
    print("Different First Letter")
```

```
list1 = [0] * 10
set1 = set({})
```

```
for i in list1:
    set1.add(i)

print(len(set1))
```

```
string1 = ['This', 'is', 'a', 'short', 'phr']
string2 = ['This', 'is', 'actually', 'a', '']
```

```
string1.reverse()
string2.reverse()
print(string1)
print(string2)
```

```
plates = ["G06 WTR", "WL11 WFL", "QW68 PQR"]
plate1 = []
plate2 = []
plate3 = []
nestedPlates = [plate1, plate2, plate3]
```

```
for i in plates[0]:
    plate1.append(i)
```

```
for i in plates[1]:
    plate2.append(i)
```

```
for i in plates[2]:
    plate3.append(i)
```

```
year = [int(plates[0][1])+int(plates[0][2])
```

```
print(type(year[0]), type(year[1]), type(year[2]))
print((year[0]), (year[1]), (year[2]))
```

```
# FIND UNIQUE VALUES IN A LIST

my_list = [1, 2, 3, 3, 3, 4, 5]

def unique(original_list):
    unique_list = []

    for x in original_list:
        if x not in unique_list:
            unique_list.append(x)

    if unique_list == 0:
        print("All elements are unique")
    else:
        print("There are duplicate elements")
```