Python

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

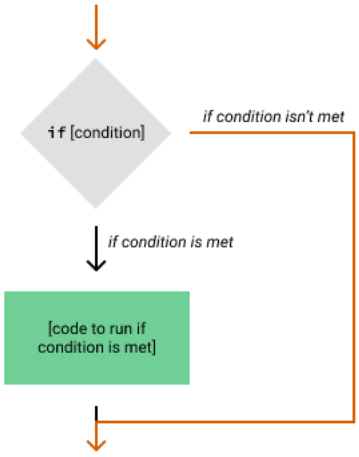
- The backslash (\) character is utilised to escape apostrophes or other special characters in strings.
- Strings are iterable, meaning that they can return their elements one at a time
- Strings are also immutable, meaning that their elements cannot be changed once assigned. Must be **REASSIGNED** to 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

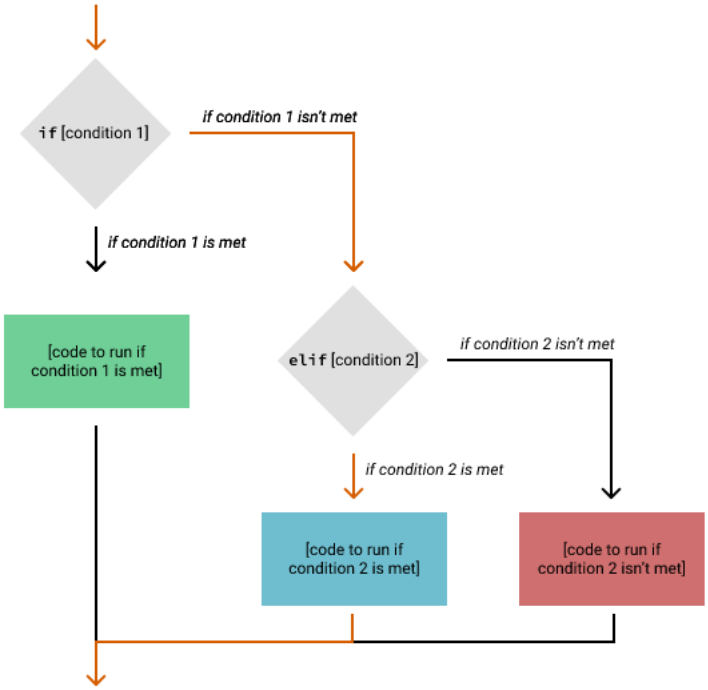
Conditional Statements

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")
```

Functions

Len()

```
# Get the length of a string
word = "Hello World"
len("Hello World")
len(word)
```

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}")
```

Replace()

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

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 {:1.3f}".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 {:1.3f}".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))

```
# Transorms 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.

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

- 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 docstringcgs 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
```

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 inte
                    b (int): Another decima

            Returns:
                    binary_sum (str): Binar
    '''
    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 attrib

        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 = statistice.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)
```

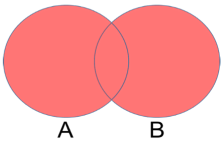
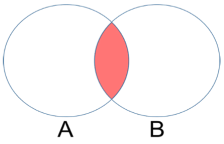
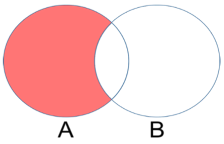
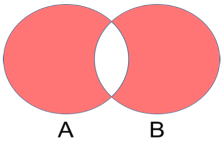
```
long_word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'
print(long_word)

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 alphebetically
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(srt_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 common
final_set = set1.difference(set2) # Returns a set with the items in set1 but not set2
final_set = set1.symmetruc_difference(set2) # Returns a set with items in set1 and set2, removing the common 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 simpleAddiction(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'''
xrange() '''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'''
```

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

print('Hello', x)
```

Modules

```
pi@raspberrypi ~ $ sudo apt-get install python-matplotl:
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(ye
print((year[0]), (year[1]), (year[2]))
```