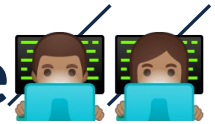
Python fundamentals

Welcome 

Getting started with Python

Damilola Omifare
Software Engineer

Class Ethics

Please!!!

- No phones unless it is very important.
- No talking please.
- Stop me if you have questions or doubts (*no question is stupid*)
- Have fun learning

Scope

This session is designed to imparting a basic level understanding of variables, control flow, functions in python programming.

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.

```
x = 5
```

```
y = "John"
```

Python allows you to assign value to multiple variables in one line.

e.g `x, y, z = "Orange", "Banana", "Cherry"`

They are reserved words.

Keywords in Python programming language

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Data Types

- Integer **1,2,3,4,5**
- floats : 1.2,1.4, 53.5
- complex
- strings : `"Hello World!"`
- booleans : true / false. 1 or 0
- bytes, bytearray, memoryview
- ...

Getting the Data Type: You can get the datatype of any object by using the type() function:

Strings

In python, strings are shown as variable type **str**. You can define a string with either double quotes " or single quotes '.

```
>>> my_string = 'this is a string!'

>>> my_string2 = "this is also a string!!!"

>>> this_string = 'David\'s laptop is a macbook'
```

Strings formatting

In python, these are methods you can use on strings.

capitalize()	encode()	format()	isalpha()	islower()	istitle()
casefold()	endswith()	format_map()	isdecimal()	isnumeric()	isupper()
center()	expandtabs()	index()	isdigit()	isprintable()	join()
count()	find()	isalnum()	isidentifier()	isspace()	ljust()

Python also has : .**split(), .format()** methods.

Data Structures

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.

List

- A list is a container organising data which is ordered and changeable. In Python lists are written with square brackets.
- A list is one of the most common and basic data structures in Python.

```
mylist = ["hello", True, 1, 4.3]
print(mylist)
```

Accessing a list :
You access the list items by referring to the index number.
The number could be positive or negative.

```
print(mylist[0]) // hello
print(mylist[1]) // True
print(mylist[-1]) // 4.3
```

Slicing and dicing with List

- When using slicing, it is important to remember that the `lower` index is **`inclusive`** and the `upper` index is **`exclusive`**.

```
mylist = ["hello", True, 1, 4.3]
print([0:1]) // Expected output is hello
```

```
print(mylist[len(mylist)]) // throws an error
print(mylist[len(mylist)] - 1) //  4.3
```

List method

- `len()` returns how many elements are in a list.
- `max()` returns the greatest element of the list. How the greatest element is
- `min()` returns the smallest element in a list. min is the opposite of max, which returns the largest element in a list.
- `sorted()` returns a copy of a list in order from smallest to largest, leaving the list unchanged.

These operations can be performed on a list: `join, append, pop, count, reverse, copy, clear, remove, delete` …

`From more visit :`

https://docs.python.org/3/tutorial/datastructures.html

Loop through a list

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

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

Python list comprehension.

- List comprehensions provide a concise way to create lists

```
mylist = ["apple", "python", "banana", "cherry"]
print([i for i in mylist])
```

Tuple

- It's a data type, **immutable** ordered sequences of elements.
- Elements can be mixed.

```
dimension = 52, 40, 100
length, width, height = dimension

print("The dimension are {} x {} x {}".format(length, width, height))
length, width = size

print("The size are {} x {}".format(length, width))
```

The parentheses are optional when defining tuples. It is know that programmers do frequently omit them if parentheses don't clarify the code.

Tuple

- provides a convenient way to **swap** variable values.
- used to **return more than one value** from a function
- You can **iterate** over a tuple

```
(a ,b) = (b, a)
```

```
a = b
b = a
```

```
hello = a
a = b
b = hello
```

```
def multipy and_addition (x, y):
    add = x + y
    multipy = x * y
    return (add, multipy)

(added, multiplied) =
multipy_and_addition( 2,6)
```

return more than one value

Set

- A **set** is a data type, **mutable** unordered collections of **unique** elements.
- No duplicate members

```
numbers = [1,4, 4, 2, 6, 3, 1, 1, 6]
get_unique_number = set(numbers)

print(get_unique_number)



fruit = {"apple", "banana", "mango", "grapes", "watermelon"}

print(fruit)
```

More on Strings

- can **number, letter, spaces, special characters** like @
- enclose in **single or double quotation mark**. Never mixed them up

```
hi ="Hi there."
```

```
hi ='Hi there."
```

DON'T DO THIS

- **string concatenation**

```
her_name = "Pythoniana"
salute = "Hi there."
greetings = her_name + " " + salute
```

- **Operation** can be done on **strings** like, `her_name * 3`

INPUT : `input("")`

- type a description inside the quotation marks as this is printed.
- user types something and hit enter
- bind input to a variable / assign it to a variable.
- input **takes in the values as string**, so it must be casted when **working with floats, or integers (numbers)**

```
user_input = input("What is your name...")
print(user_input, "Hi there.")
```


= is not ==

NOTE!!!

= is not the same as ==

- suppose **a** and **b** are variables names
- **=** is for assigning values ,
- **==** is for comparing if two variable are the same. For instance a == b,

Comparison Operators

- suppose **a** and **b** are variables names
- the variables can **strings**, **int**, **floats**
- comparison evaluates to a **boolean, True or False**

```
a > b
a < b
a >= b
a <= b
a == b ## this is a equality check, evaluates to True if they
are the same, otherwise False
b != a ## this is a equality check, evaluates to True if they
are not the same, otherwise False
```

Logic operators

- suppose **a** and **b** are variables names
- the variables can **strings**, **int**, **floats**
- comparison evaluates to a **boolean, True or False**

```
a or b  True if either or both are True
a and b  True if both are True
not b  True if b is False, False is b is True
not a  True if a is False, False is a is True
```

Exercise

Time to take your skills for a spin 💪🏾

- suppose **a** and **b** are variables names,
- draw a table showing all possible evaluation of **a**, **b**

3 mins

Example

a	b	a and b	a or b
`True`	`True`	`True`	`True`

Pseudocode is important

Pseudocode

Algorithm are designed using pseudocode

is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading

Wikipedia

Pseudocode sample

```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
       C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```


Pseudocode & datatype exercise

Example 1: Write pseudo code that reads two numbers and multiplies them together and print out their product.
Example 2: Write pseudo code that tells a user that the number they entered is not a 5 or a 6.
Example 3: Write pseudo code that performs the following: Ask a user to enter a number. If the number is between 0 and 10, write the word blue. If the number is between 10 and 20, write the word red. if the number is between 20 and 30, write the word green. If it is any other number, write that it is not a correct color option.
Example 4: Write pseudo code to print all multiples of 5 between 1 and 100 (including both 1 and 100).
Example 5: Write pseudo code that will count all the even numbers up to a user defined stopping point.
Example 6: Write pseudo code that will perform the following.

a) Read in 5 separate numbers.
b) Calculate the average of the five numbers.
c) Find the smallest (minimum) and largest (maximum) of the five entered numbers.
d) Write out the results found from steps b and c with a message describing what they are

Homework 1: Write pseudo code that reads in three numbers and writes them all in sorted order.
Homework 2: Write pseudo code that will calculate a running sum. A user will enter numbers that will be added to the sum and when a negative number is encountered, stop adding numbers and write out the final result.

SOLUTION

Coffee Break
15m ☕

Up next

PROGRAM FLOW

Exercise

Time to take your skills for a spin 💪🏾

- suppose **a** and **b** are variables names,
- draw a table showing all possible evaluation of **a**, **b**

3 mins

Example

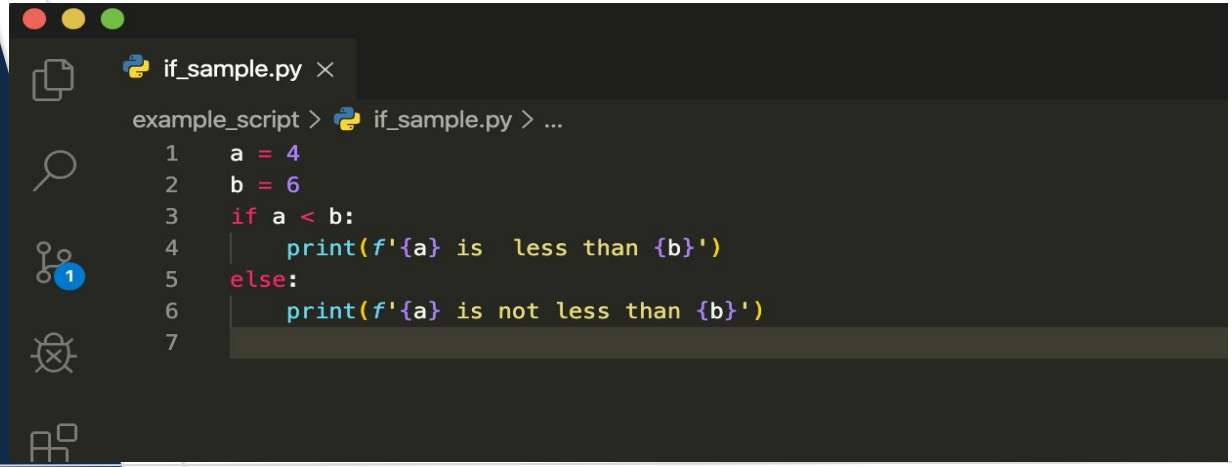
a	b	a and b	a or b
`True`	`True`	`True`	`True`

If

if statement is a conditional statement that runs or skips code based on whether a condition is **True** or **False**.

```
if <condition>:
    <expression>
    <expression>
    ...
```

evaluates the code block if **True**



elif

elif is used to check for an additional **condition** if the conditions in the previous clause(**s**) in the **if** statement evaluate to **False**.

```
if <condition>:
    <expression>
    ...
if <condition>:
    <expression>
    …
elif <condition>:
    <expression>
    …
...
```

If it evaluates to **False**

This condition is evaluated.

elif

example_script > elif_else_example.py > ...

```
1   a = 4
2   b = 6
3   c = 4
4   if a < b:
5       print(f'{a} is  less than {b}')
6   if c == a:
7       # if true 'Hurray' will be printed,
8       # if False, it goes into the elif below.
9       # Try changing the conditon to evaluate to false.
10      print(f'Hurray')
11  elif type(a) is type(b):
12      print(f'{a} and {b} are of the same type ')
13  else:
14      print(f'{a} is not less than {b}')
15
```

else

- **else** is the last clause after an **if** clause , if used.
- Code in the **else** runs if all conditions above in that in the if statement evaluate to False.
- Note else doesn't require a condition.

Exercise : Class grade

10 mins

- Write a python program that prints the grade of the student based on the score.

In your program, the user should be able to input their name and grades. if user input 75, it should print : [User's name] passed with a grade of ***A***

Marks	Grade
75 - 100	**A**
65 - 74	**B**
55 - 64	**C**
45 - 54	**D**
0 - 44	**Fail**

Coffee Break
15m ☕

Up next

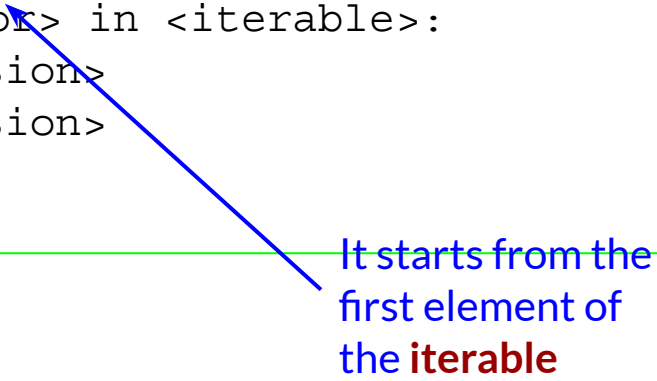
PROGRAM FLOW

Loops

For loop: for

- **for** keyword is used to do something **repeatedly** (**iterate**) over an **iterable**.
- **iterable** is an object that can return one of its element at a time.
- **iterable** could be strings, lists, tuples and other non sequential data structures.

```
for <iterator> in <iterable>:
    <expression>
    <expression>
    ...
```



For loop: example

example_script > for_loop.py > ...

```
1  my_list = [
2      6, "Lisbon", "new york",
3      ["This is now", "Hello", 123],
4      2, 3, 4]
5  for i in (my_list):
6      # i is the iterator
7      # my_list is the iterable.
8      # 6 is the first element and expected to be printed first.
9      print(i)
```

range() in for loop

- **range (start, stop, step)**
- default values : **start** is 0, **step** = 1 and optional
- keeps the loop until **step - 1**

```
for <iterator> in range(start, <iterable>, step):
    <expression>
    <expression>
    ...
```

When given it specifies the increment

range() example

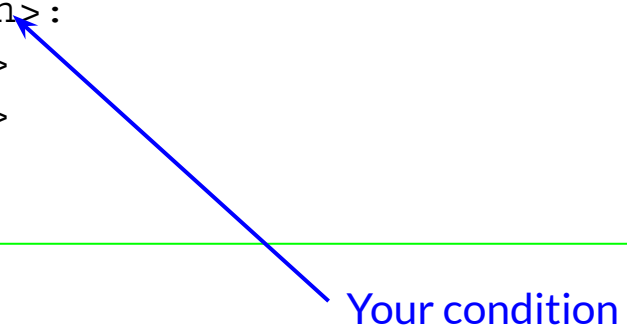
example_script > range_example.py > ...

```
1    for i in range(2, 7, 2):
2        print(i)
3
4
```

while loop

- **while** keyword is used to do something **repeatedly** until a condition is met. For short, it checks the condition again until it is met.
- condition evaluates to **boolean** i.e True or False
- loop can run forever if you do not specify when to exit.

```
while <condition>:
    <expression>
    <expression>
    ...
```



while example

example > while_example.py > ...

```
1    a = 1
2    while a < 4:
3      print(a)
4      a += 1
```

comparison btw for and while

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

Source **: MIT**

Break / Continue

break is used to break out of a loop
continue skips one iteration of a loop

zip

- The **zip()** returns an iterator that combines multiple **iterables** into one sequence of tuples.
- Each tuple contains the elements in that position from all the **iterables**.

zip_example.py

example > zip_example.py > ...

```
1   a = ("Chelsea", "Manchester", "Chester")
2   b = ("Rain", "Snow", "Winter")
3
4   x = zip(a, b)
5   #use the tuple() function to display a readable version of the result:
6   print(tuple(x))
7   # Output : (('Chelsea', 'Rain'), ('Manchester', 'Snow'), ('Chester', 'Winter'))
8
9   for i, j in zip(a, b):
10      print("{}: {}".format(i, j))
11  # Output :
12  # Chelsea: Rain
13  # Manchester: Snow
14  # Chester: Winter
15
16
```

enumerate

- **enumerate** is a built in python function.
- it returns an iterator of tuples containing indices and values of a list
- it allows you to have index along with each element of an **iterable** in a **loop**.

example > enumerate_example.py > ...

```
1   seasons = ['raining', 'snowing', 'autumn', 'spring', 'fall', 'summer']
2   for i, season in enumerate(seasons):
3       print(i, season)
4   # Expected Output :
5   # 0 raining
6   # 1 snowing
7   # 2 autumn
8   # 3 spring
9   # 4 fall
10  # 5 summer
11
```

Function

- **function** keyword is a reusable pieces of code.
- Functions are designed to achieve a specific task.
- Functions are not run until they are called or invoked.

function properties are :

- declared with the keyword **def**
- has a **name**
- has **parameters** (0 or more)
- has a **docstring** (optional but recommended best practice)
- has a body
- **return** something

Function example

```
def <function_name>(parameters):
    """What is your name..."""  # docstrings
    <expression> # function body
    <expression>
    ...
    return something
function_name(its_parameters) # If any
```


Function sample

```
def multipy and_addition (x, y):
    add = x + y
    multiply = x * y
    return (add, multiply)

(added, multiplied) = multipy_and_addition( 2,6)
```


Thank you for your time

Up next :

Project 1

Mad Lib Generator