Hints for Python. **Presented by/ Eng. Mostafa Atlam**
Facebook Page: https://www.facebook.com/profile.php?id=61554255886092

print("Hello, World!")

Python Indentation

Indentation refers to the spaces at the beginning of a code line (in other programming languages the indentation in code is for readability only).

Python uses indentation to indicate a block of code.

Example:

```
if 5 > 2:
  print("Five is greater than two!")
```

Comments in python:

→ is used for a single line comment.

There is no multi-line comment in python, so we use **"""Comment"""** that is a string but not assigned to a variable.

Variables:

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

```
x = 5
y = "John"
x = 4          # x is of type int
x = "Sally" # x is now of type str
```

Casting:

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

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

You can get the data type of a variable with the **type()** function.

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

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

A variable name cannot start with a number.

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).

Assigning multiple values:

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

x = 5
y = 10
print(x + y) #This will work as operator, will print 15.

x = 5
y = "John"
print(x + y) #Raise an error.
```

If we used the **global** keyword, the variable belongs to the global scope:

```
global x
```

The three numeric types in Python: **int**, **float** and **complex**

Strings

A string can be defined using **"", """ """** and **' '.**

```
a = """Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididuntut
labore et dolore magna aliqua."""
print(a); raw_s = r'Hi\nHello'; s = 'Hi\nHello' (diff)
print(a[1]) # This will print L
```

We can loop through a string:

```
for x in "banana":
  print(x) # This will print each char in a line.
```

The **len()** function returns the length of a string.

To check if a certain phrase or character is present in a string, we can use the keyword in:

```
txt = "The best things in life are free!"
print("free" in txt)
```

To check if a sub-string is not in a string, we can use **not in.**

```
print(a[2:5]) # The start will be 2 and the last address
              # is excluded which is 5.
print(a[:5])  # Get the characters from the start to
              # position 5.
print(a[2:]) # Get the characters from position 2, and
             # all the way to the end.
print(a[-1]) # This will print the last char in the str.
print(a[::-1]) # This will print the string but reserved.

a = "Hello, World!"
print(a.upper()) # HELLO, WORLD!
print(a.lower()) # hello, world!
print(a.capetalize()) # Hello, world!
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

The **replace()** method replaces a string with another string. **a.replace("H", "J", 2) # 2 no. of occurrences.**

The **split()** method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']

str1 = str2 + " " + str3 # Concatenation
str1 = "Mostafa"; str2 = "Samy"
str3 = f"My name is {str1} {str2}" # Formatted String.

txt = "We are the so-called \"Vikings\" from the
north." # Escape Character
```

Membership Operators:

in: Returns True if a sequence with the specified value is present in the object.

not in: is the opposite.

Bite-wise operations:

& → AND → Sets each bit to 1 if both bits are 1.

| → OR → Sets each bit to 1 if one of two bits is 1.

^→XOR→Sets each bit to 1 if only one of two bits is 1.

~ → NOT → Inverts all the bits.

<< Zero → fill left shift → Shift left by pushing zeros in from the right and let the leftmost bits fall off.
>> → Signed right shift → Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.

These operators are unlike logical operators and, or and not.
Precedence:
() → ** → ~ → *, /, //, % → +, - → <<, >> → & → ^ → | → ==, !=, >, >=, <, <=, is, is not, in, not in → not → and → or.

Lists:

A collection of data items. Mutable, allow duplicates and a list can be of different data types.
To print the number of items of a list:

```
thislist = ["apple", "banana", "cherry", "orange"]
print(len(thislist))
thislist[-1] → # ["orange"]
thislist[1] → # ["banana"]
thislist[1:3] → # ["banana", "cherry"]
thislist[:3] → # ??? and thislist[2:] → ???
```

To know if a specific item in a list, we can say:

```
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

To change a specific item in the list (for ex/ second):

```
thislist[1] = "blackcurrant"
```

To change a range of items in the list:

```
thislist[1:3] = ["blackcurrant", "watermelon"]
```

Insert "watermelon" as the third item:

```
thislist.insert(2, "watermelon")
```

To add at the end of the list, use the **append()** method.
To extend a list with another list use the **extend()** method.

```
list1 = ["mango", "pineapple", "papaya"]
thislist.extend(list1)
# equivalent to: thislist  = thislist + tropical
```

pop(index) and **remove(element_value)** methods can be used to remove elements from a list. The main difference is that **pop()** returns the removed element.

```
Thislist.clear() # will return an empty list
```

Loop through lists:

```
thislist = ["apple", "banana", "strawberry"]
for x in thislist:
  print(x) # This is equivalent to

for i in range(len(thislist)):
   print(thislist[i])
```

List Comprehension offers the shortest syntax for looping through lists:

```
thislist = ["apple", "banana", " strawberry"]
newlist = [x for x in thislist if x != "apple"]
# Create a new list from fruits list but escape apple.
newlist = [x for x in range(10) if x < 5]
thislist.sort() # sort in ascending order
thislist.sort(reverse = True) # descending
list = thislist
# any modifications in list will appear in thislist and the
# opposite.
```

To avoid this:

```
list = thislist[:] # This will create a new memory ref.
```

Tuples:

Tuples are used to store multiple items in a single variable. A tuple is a collection which is ordered and **unchangeable (immutable)**. Tuples are written with round brackets. Tuples allow duplicates.

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple)) # Print the length of the Tuple
```

To create a tuple with only one element:

```
thistuple = ("apple",)
```

To access the elements of a tuble:

```
thistuple[-1] # cherry
thistuple[1:] # Start from element to till the last.
thistuple[:2] # Start from the first element to
              # element 1.
```

We can update a tuple as following:

```
# Convert the tuple into a list to be able to change
# it:
x = ("apple", "banana", " grapes")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
```

To add elements to the tuple:

```
# Convert the tuple into a list, add "orange", and
# convert it back into a tuple:
thistuple = ("apple", "banana", " grapes")
y = list(thistuple)
y.append("orange")
```

To extract the values back into variables, this is called "unpacking":

```
fruits = ("apple", " grapes")
(a, b, c) = fruits
print(a)
print(b)
```

Methods for tuples:

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

Dictionary:

It consists of a key and a value. It is defined using {}.

```
students = {
   "name": "Ali",
   "age": 20,
   "emails": [ "m@m.com", "m@a.com" ],
   "address": {
      "street": "ABCD",
      "flat": 15
   } }
```

To access elements of the dictionary:

```
print(students["name"])
print(students["emails"][0])
print(students["address"]["street"])
# Change the value for key “name” from Mostafa to
# Ahmed.
students["name"] = "Ahmed"
# If the key does not exist it will append this key in
# the dictionary and set its value.
```

In case of trying to retrieve a key that is not found in the dictionary, this will raise an error. To solve this problem:

```
if "abc" in students:
  print(students["abc"])
# Another and better way:
print(students.get("abc"))
# This will search for the key, if it is found will
# return the value and if not it will return None.
print(students.get("abc", "Ali")) # If not return Ali.
```

To loop through keys:

```
for key in students: # The default here that it will
                     # pass through the keys.
  print(key)
# To loop through values, we can loop through
# students.values()
```

To remove the value assigned to a specific key:

```
students["key_name"] = None
```

To remove the key itself:

```
del(students["key_name"])
```

To remove the key and return its value:

```
students.pop("key_name")
```

To loop through both keys and values:

```
for key, value in students.items():
#Each time this loop returns a tuple of each key and
# its value
  print (f"{key} => {value}")
```

To update keys and values of a dictionary, use update function:

```
students.update({"name": "Ali", "age": 20})
# We can send a dictionary of keys and values we
# want to update to this function.
```

```
def get_sum(x, y, z = 3): # give a default value for z.
  return x + y + z
```

There are two methods of calling:

- Positional calling:

```
print(get_sum(5, 6, 7))
```

- Calling by input parameters name:

```
print(get_sum(x=5, y=6, z=7))
```

```
def get_sum(x, y, *args): # args is a naming
                          # convention
  print(x)
  print(y)
  print(args) # This means zero or unlimited
              # number of arguments (positional).
              # It will create a tuple of these
              # elements.
  print(type(args)) # Tuple

def get_sum(x, y, **kwargs): # kwargs is a naming
                             # convention
  print(x)
  print(y)
  print(kwargs)      # This means zero or unlimited
                     # number of arguments
                     # (named arguements).
  print(type(kwargs)) # This will be a dictionary of
                      # the named parameters as
                      # keys.
print(get_sum(x = 1, y = 2,  a = 3, b = 4, c = 5))
# This will create a dictionary of a, b, and c as keys
# and 3, 4, and 5 are the values.
```