Module02_Day01_PyRefresher_1

December 16, 2022

1 Python Refresher 1

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

```
Hello
```

[ ]:
```
print("Hello World!")
```

```
Hello World!
```

[ ]:
```
print(1+4)
```

```
5
```

1.0.1 Data Types

Primitive Data Type: In programming, a primitive data type is a data type that is not derived from any other data type. It is the most basic type of data that a programming language can manipulate.

F.C.C. [First Class Citizens] = A first-class citizen (also known as a first-class object) in programming is a value, function, or data type that can be treated in the same way as any other value, function, or data type. This means that a first-class citizen can be passed as an argument to a function, returned as a result from a function, assigned to a variable, or stored in a data structure, just like any other value.

- Everything in Python is an object

Variables In programming, a variable is a named location in a computer's memory where a programmer can store and retrieve data. A variable is like a container that can hold a value, and the value can be changed or updated as needed during the execution of a program.

- Placeholders
- Containers

DocStrings

[ ]:
```
def myfunc():
    """a=5 return a""" # Docstrings

print(myfunc.__doc__)
```

```
print(myfunc)
```

```
a=5 return a
<function myfunc at 0x000001D2845F41F0>
```

[ ]:
```
a = 5
```

[ ]:
```
print(a)
```

```
5
```

[ ]:
```
print("a")
```

```
a
```

[ ]:
```
# Why there is no Out section above ^ & it is below
# Because print() is actual Output and all else are temporarily Output
```

[ ]:
```
type(a)
```

[ ]:
```
int
```

[ ]:
```
print(type(a)) # So everything in Python is Object
```

```
<class 'int'>
```

---

isinstance

[ ]:
```
isinstance(a, int)
```

[ ]:
```
True
```

[ ]:
```
isinstance(15, object) # Everything is object
```

[ ]:
```
True
```

[ ]:
```
isinstance("Hello", object) # Everything is object
```

[ ]:
```
True
```

[ ]:
```
isinstance(type, object) # Even functions are object
```

[ ]:
```
True
```

[ ]:
```
isinstance("""Hello""", object) # Even docstrings are object
```

[ ]:
```
True
```

1.0.2 Language Type

Dynamic Type Language: Languages where you don't require defining containers for variable

Statically Typed Language: Languages that need containers/data type for variables like int, float etc

[ ]:
```
x = 5
x, type(x)
```

[ ]:
```
(5, int)
```

[ ]:
```
x ="Hello"
x, type(x)
```

[ ]:
```
('Hello', str)
```

[ ]:
```
x = 10
print(x)
```

```
10
```

[ ]:
```
type(x)
```

[ ]:
```
int
```

[ ]:
```
id(x)
```

[ ]:
```
2005743135312
```

[ ]:
```
x = "Hello"
```

[ ]:
```
print(x)
```

```
Hello
```

[ ]:
```
type(x)
```

[ ]:
```
str
```

[ ]:
```
id(x) #ID doesnot match so it is a Dynamically Typed Language
```

[ ]:
```
2003676086064
```

1.0.3 Memory Workflow

[ ]:
```
alpha = 578
beta = 989

print(id(alpha), id(beta))
```

```
2003710409040 2003710409264
```

```
[ ]: alpha = beta
print(id(alpha), id(beta))
```

```
2005743135152 2005743135152
```

```
[ ]: alpha = 578 # Still different
beta = 578

print(id(alpha), id(beta))
```

```
2003710409296 2003710409328
```

Small Integer Caching/ Integer Interning Range: -5, 256

```
[ ]: # Beforehand Python has created small integers in memory from range(-5,256)

alpha = 5 # But same now, Integer Interning
beta = 5

print(id(alpha), id(beta))
```

```
2005743135152 2005743135152
```

```
[ ]: x = 5
id(x)
```

```
[ ]: 2005743135152
```

```
[ ]: x = "hello"
id(x)
```

```
[ ]: 2003671713648
```

x points to 5 and after that it points to other location, but 5 is still there in RAM, so **Python Garbage Collector** comes and clear the variable.

1.0.4 Identifiers

In programming, an identifier is a name given to a variable, function, or other element in a program. An identifier is used to refer to the element in the program, and it must follow the rules for naming identifiers that are defined by the programming language.

A variable, on the other hand, is a named location in a computer's memory where a programmer can store and retrieve data. A variable has a name, a data type, and a value. The name of the variable is the identifier that is used to refer to it in the program, and the data type specifies the type of data that the variable can hold. The value is the data that is stored in the variable.

Identifiers: Name given to a variable, function, object, classes etc

3 rules as identifier:- - Starts with *(a-z, A-Z or _) - *Inbetween you can use anything other than alphanumeric &* (a-z, A-Z, 0-9 or_)_ - Should not be a python reserved keywords

[ ]:
```
import keyword

print(keyword.kwlist)
```

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

[ ]:
```
int = 4 # Not Recommended, int is not a reserved keyword, It is a class
```

[ ]:
```
int
```

[ ]: 4

[ ]:
```
int("5") # Error because int has been removed as a class and now used as an␣
↪identifier
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_21608\4289630484.py in <module>
----> 1 int("5")

TypeError: 'int' object is not callable
```

[ ]:
```
del int # if assigned use del
```

[ ]:
```
int("5")
```

[ ]: 5

1.0.5 Mutability/Immutability

Mutability If a data type values can be changed/edited without changing there location - Set,List & Dictionary - only these 3 are mutable in Python

[ ]:
```
x = 1000
id(x)
```

[ ]: 2003710409264

[ ]:
```
x = x + 1
id(x)
```

[ ]:
```
2003710409360
```

Hence Integer is not Mutable, because it creates new Values at a different location and points to it

Immutability‘ If a data type values cannot be changed/edited without changing there location - Like: int,str,tuple etc

1.0.6 Operators

Arithematic Operators

+,-,*,/,//,**<%

/ => is a float division

[ ]:
```
10/2
```

[ ]:
```
5.0
```

// => is a floor division

[ ]:
```
# Floor devision
8//3
```

[ ]:
```
2
```

** is power

[ ]:
```
2**3
```

[ ]:
```
8
```

% gives remainder

[ ]:
```
10 % 3
```

[ ]:
```
1
```

Logical Operators

or, and not

[ ]:
```
not True
```

[ ]:
```
False
```

[ ]:
```
not False
```

[ ]:
```
True
```

Conditional operators

==, >, <, >=, <=,!=

1.0.7 Control Flows

Conditional Statements

[ ]:
```
age = 21

if age>=18:
    print("Eligible to vote")
```

```
Eligible to vote
```

[ ]:
```
age = 15

if age>=18:
    print("Eligible to vote")
```

[ ]:
```
age = 15

if age>=18:
    print("Eligible to vote")
else:
    print("Go home Kid!!!")
```

```
Go home Kid!!!
```

```
#10L > 30% Tax
# 5L -  20% Tax
# < 5L - 0 Tax
```

[ ]:
```
salary = 4
tax = 0

if salary > 10:
    tax = 0.3 * salary
else:
    if salary > 4:
        tax = 0.2 * salary
    else:
        tax = 0
print(tax)
```

```
0
```

[ ]:
```
salary = 4
tax = 0

if salary > 10:
```

```
    tax = 0.3 * salary
elif salary > 4:
    tax = 0.2 * salary
else:
    tax = 0
print(tax)
```

```
0
```

1.0.8 Loops

- while
- loop

While

- When you know the exit condition

[ ]:
```
i = 0

#stopping condition
while i<=10:
    print(i,end=" ")

    #updation
    i = i+1
```

```
0 1 2 3 4 5 6 7 8 9 10
```

[ ]:
```
command = input("Enter Command:")

while command!="exit":
    print(command)
    command = input("Enter Command:")
```

```
Enter Command:h
h
Enter Command:s
s
Enter Command:a
a
Enter Command:exit
```

[ ]:
```
# Describe what function does, inbuilt jupyter feature not python
```

[ ]:
```
print?
```

Loop

- When you know the start and end condition

[ ]:
```
for i in range(1,11): #start is optional and by default it is 0
    print(i, end=" ")
```

```
1 2 3 4 5 6 7 8 9 10
```

[ ]:
```
for i in range(10,0,-1): #start is optional and by default it is 0
    print(i,end=" ")
```

```
10 9 8 7 6 5 4 3 2 1
```

[ ]:
```
for i in range(0,21,2): #start is optional and by default it is 0
    print(i,end=" ")
```

```
0 2 4 6 8 10 12 14 16 18 20
```

[ ]:
```
n = 5
for i in range(1,n+1):
    for j in range(i):
        print("*",end=" ")
    print()
```

```
*
* *
* * *
* * * *
* * * * *
```

[ ]:
```
n = 5
for i in range(1,n+1):
    print("* "*i)
```

```
*
* *
* * *
* * * *
* * * * *
```

1.0.9 Doubts Notes

Dry Run: Is running which code manually and telling which lines will be executed

[ ]:
```
s1 ="Hello World!!!!"
s2 = "mohit.uniyal@scaler.com"
id(s1),id(s2)
```

[ ]:
```
(2003709641584, 2003713389952)
```

[ ]:
```
s1 = "Scaler"
s2 = "Scaler"
```

```
id(s1), id(s2)
# This happens because of Caching but this is not String Interning.
# There ids could be same if they have same value and no special characters␣
 ↪including space
```

[ ]: (2003705499824, 2003705499824)