Print Arguments, Enum & Eval, Memory & Pointers, Jupyter walk-through & Magic methods

Print Arguments

As you already know, we can print data on the terminal in Python by simply using the print statement. But we saw that by default, each print statement prints text in a new line.

```
print("Hello")
print("World")
```

```
Hello
World
```

What if we want to print in the same line using separate/different print statements?

`end` parameter

Python's `print()` function comes with a **parameter called `end`**. By default, the value of this parameter is `\n`, i.e., the **new line character**.

You can end a print statement with any character/string using this parameter.

```
print("Hello", end="")
print("World")

print("Hello", end=" ")
print("World")
```

```
HelloWorld
Hello World
```

The **value of `end` is appended to the output** and the **next print continues from there**.

```
print("Welcome to" , end = " ")
print("Scaler", end = " ")
```

```
Welcome to Scaler
```

```
# ends the output with a different character '@'
print("Python" , end = '@')
print("Scaler")
```

```
Python@Scaler
```

`sep` parameter

The separator between multiple arguments to `print()` function in Python is space by default, also known as the **softspace feature**.

```
print('Scaler','for', 'Python', 2022)
```

```
Scaler for Python 2022
```

This can also be modified and can be made to any character, integer or string as per our choice. The **`sep` parameter is used to achieve the same**.

```
#code for disabling the softspace feature
print('Scaler','For','Python', sep='')
```

```
ScalerForPython
```

It is also used for formatting the output strings.

```
#for formatting a date
print('24','02','2022', sep='-')
```

```
09-12-2016
```

```
#another example
print('Python','Scaler', sep='@')
```

```
Python@Scaler
```

The `sep` **parameter when used with the** `end` **parameter** can produces awesome results.

Some examples by combining the `sep` and `end` parameters:

```
print('Scaler','For', 'Python', sep='', end='')
print('Scaler')
#\n provides new line after printing the year
print('24','02','2022', sep='-', end='\n')

print('Python', end='@')
print('Scaler','Academy', sep='')
```

```
ScalerForPythonScaler
24-02-2022
Python@ScalerAcademy
```

Enumerate and Eval

`enumerate`

Have you ever needed to loop over a list and also needed to know where in the list you were at?

Often, when dealing with iterators, you also might need to keep a count of iterations. One thing you could do is you could add a counter that you increment as you loop:

```
my_string = 'abcdefg'
counter = 0
for letter in my_string:
    print (counter, letter)
    counter += 1
```

```
0 a
1 b
2 c
3 d
4 e
5 f
6 g
```

As a more simple solution you could use Python's **built-in `enumerate()` function**! Python eases the task by providing `enumerate()` to keep track of count of iteration automatically.

`enumerate()` method adds a counter to an iterable and returns it in a form of enumerating object.

Let's try it out on a list of strings!

```
l1 = ["eat","sleep","repeat"]

# creating enumerate objects
obj1 = enumerate(l1)

print ("Return type:",type(obj1))
print (list(enumerate(l1)))
```

```
Return type: <class 'enumerate'>
[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]
```

We can also change the start index with `enumerate()`

```
# changing start index to 2 from 0
s1 = "scaler"
obj2 = enumerate(s1)
print (list(enumerate(s1,2)))
```

```
[(2, 's'), (3, 'c'), (4, 'a'), (5, 'l'), (6, 'e'), (7, 'r')]
```

This enumerated object can be used directly for loops or **converted into a list of tuples** using the list() method.

```
l1 = ["eat","sleep","repeat"]

# printing the tuples in object directly
for ele in enumerate(l1):
    print(ele)
```

```
(0, 'eat')
(1, 'sleep')
(2, 'repeat')
```

```
# changing index and printing separately
for count,ele in enumerate(l1,100):
    print (count,ele)
```

```
100 eat
101 sleep
102 repeat
```

```
#getting desired output from tuple
for count,ele in enumerate(l1):
  print(count)
  print(ele)
```

```
0
eat
1
sleep
2
repeat
```

`eval`

Python `eval()` function **parse the expression argument** and **evaluate it as a python expression**.

It runs python expression (code) within the program.

Let's take a look at a simple example:

```
var = 10 # int
source = 'var * 2' # string
print (eval(source)) # eval() evaluates the string as a numeric expression
```

```
20
```

The `eval` built-in is **fairly controversial** in the Python community. The reason for this is that **eval accepts strings and basically runs them**.

If you were to allow users to input any string to be parsed and evaluated by `eval` , then you just created a **major security breach**.

However, if the code that uses eval cannot be interacted with by the user and only by the developer, then it is okay to use. Some will argue that it’s still not safe.

```
evaluate = 'x*(x+1)*(x+2)'
print(evaluate)
print(type(evaluate))

x = 3
print(x)
print(type(x))

expression = eval(evaluate)
print(expression)
print(type(expression))
```

```
x*(x+1)*(x+2)
<class 'str'>
3
<class 'int'>
60
<class 'int'>
```

Even though `eval` is not much used due to security reasons as we explained above, still, it **comes in handy in some situations** like:

- You may want to use it to **allow users to enter their own “scriptlets”**: small expressions (or even small functions), that can be used to customize the behavior of a complex system.

- `eval` is also sometimes used in applications **needing to evaluate math expressions**. This is much easier than writing an expression parser.

Memory and Pointers

Pointers don’t exist in Python as such. **The objective of pointers is met in Python with the help of objects.**

Everything is an object in Python

The following code proves that the **int, str, list, and bool data types are each objects in Python**:

`isinstance()` method gives `True` if something is an instance of a particular data type

```
print(isinstance(int, object))
print(isinstance(str, object))
print(isinstance(list, object))
print(isinstance(bool, object))
```

```
True
True
True
True
```

This proves that **everything in Python is an object**.

But what is an object?

A Python object comprises of **three parts**:

1. Reference count
2. Type
3. Value

Reference count is the **number of variables that refer to a particular memory location**.

Type refers to the **object type**. Examples of Python types include int, float, string, and boolean.

Value is the **actual value of the object that is stored in the memory**.

That’s all you need to know for the time being

There is no specific concept of pointers in Python. **Python doesn’t need pointers as every variable is a reference to an object**.

These references are slightly different from C++ references, in that **they can be assigned to** - much like pointers in C++. Python standard way of handling things supports you.

Walk-Through Jupyter Notebook

For a quick understanding of how Jupyter Notebook works and how you can use it to create and execute your programs, you can go through the following links:

1. https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html (https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html)

2. https://jupyter.org/ (https://jupyter.org/)

You should be aware of and be comfortable using basic things at the end of these readings, like:

1. How to launch Jupyter Notebook App
2. How to change Jupyter Notebook startup folder
3. How to shut down Jupyter Notebook App
4. How to close a notebook: kernel shut down
5. How to execute a notebook
6. How to work in Jupyter Notebook cells

... and some other basic things

Magic Commands in Jupyter Notebook

Magic commands are **special commands** that can help you with running and analyzing data in your notebook. They add a **special functionality that is not straight forward to achieve with python code or jupyter notebook interface**.

Magic commands are easy to spot within the code. They are either proceeded by `%` **if they are on one line of code or by** `%%` **if they are written on several lines**.

In this section, we will go through some magic commands that are used most often and see practical examples of how to take an advantage of the functionality they provide.

List all magic commands

Let's start by listing all possible magic commands that you can use in the notebook.

```
%lsmagic



Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark  %cat  %

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%js  %%

Automagic is ON, % prefix IS NOT needed for line magics.
```

If you run the line above in your notebook you should get a list similar to this. These are all the commands available to you. We will go through just a few most commonly used

ones in this section.

Run a file

You can run a python file from your jupyter notebook using this:

```
%run <file name>
```

Let's say you have a file `hello.py` which you can download from here and place it in your current working directory:

https://drive.google.com/file/d/1Voi3FsO7TUwEF0c4G3Jvjh-noFj7wdLp/view?usp=sharing (https://drive.google.com/file/d/1Voi3FsO7TUwEF0c4G3Jvjh-noFj7wdLp/view?usp=sharing)

You can run the following command in the notebook to run the file:

```
%run hello.py
```

```
Hello, world
```

Load an external file

You can load an external file to a cell by using the `%load` command.

```
%load <file_name>
```

This is a very useful command if you already have a **python file with certain functions defined and you need to use them in the current notebook**.

In order to illustrate we take an example of the file `rectangle.py` which consists of the following code:

```
def calculate_area(len, height):
    return len * height * 2
```

You can download the file from here and place it in your current working directory: https://drive.google.com/file/d/1ctpJfm1AYoBqGgO2vRUH-eAPf9usWuP8/view?usp=sharing (https://drive.google.com/file/d/1ctpJfm1AYoBqGgO2vRUH-eAPf9usWuP8/view?usp=sharing)

You can load the file content by executing the code below:

```
%load rectangle.py
```

Once you run the cell you should get the content of the file within the cell:

```
# %load rectangle.py
def calculate_area(len, height):
    return len * height
```

Now just run the cell with the loaded code and you will be **able to use the functions that were defined within the original file**. You can now **calculate the rectangle area**.

```
calculate_area(2,4)
```

```
8
```

Get an execution time

You can also time the execution of the code using the time command.

Let's generate 1,000,000 random numbers and see how long it takes:

```
%%time
import random
for i in range(0, 1000000):
    random.random()
```

```
CPU times: user 104 ms, sys: 2.47 ms, total: 106 ms
Wall time: 106 ms
```

List all variables

There is a magic command that allows you to **list all variables that are defined within the current notebook**.

It's `%who`

You can pass it a **data type after command name to list only variables of the specific data type**.

To illustrate this let's define a string variable and two int variables:

```
var_1 = 1
var_2 = 'hello'
var_3 = 100
```

Now we can list all strings with:

```
%who str
```

```
ele      evaluate        letter  my_string       s1      source  var_2
```

It also includes the strings we defined previously in this reading

We can list all integers with:

```
%who int
```

```
count    counter         expression      i       var     var_1   var_3   x
```

Get detailed information about the variable

Once we know a variable name we can inspect what are the **details of the objects that is stored in the particular variable name**. In order to do it, you can use the following command:

```
%pinfo <variable>
```

Let’s go back to the example with three variables that we used in the previous section to illustrate it better:

```
var_1 = 1
var_2 = 'hello'
var_3 = 100
```

Let’s now inspect `var_1` :

```
%pinfo var_1  # execute it and you'll see all the details about var_1
```

As you must have seen you get all the important information about the variable such as **type, string form, and docstring**.

Let's inspect a variable that is a string:

```
%pinfo var_2
```

You'll get the same information details as with an `int` and **additionally length of the string**.