CERTIFIED ASSOCIATE IN PYTHON PROGRAMMING

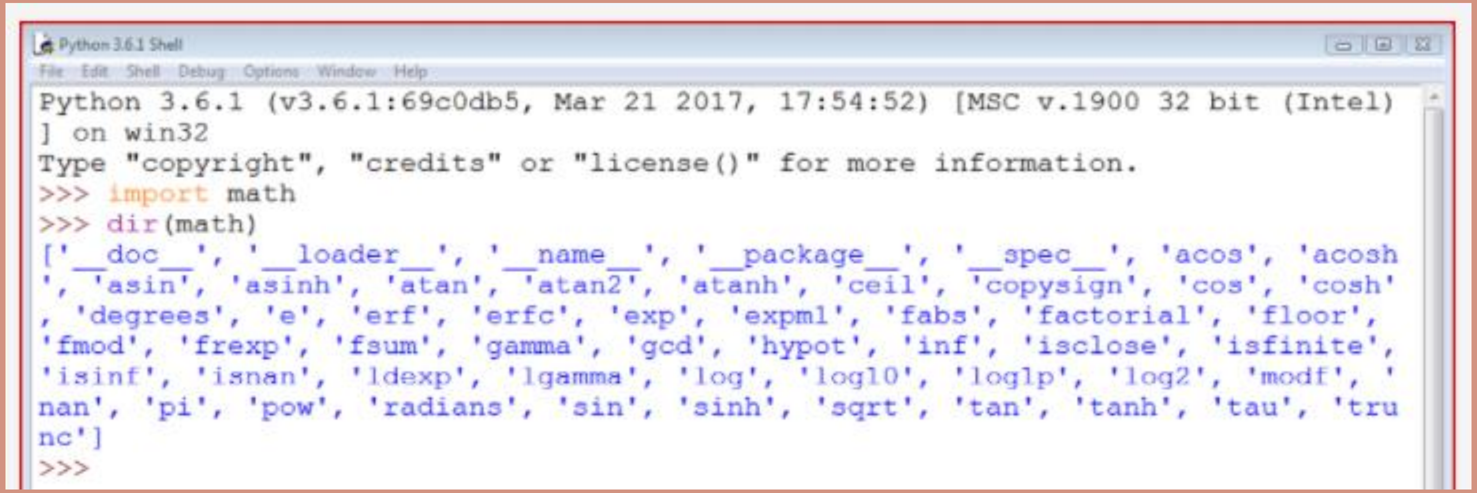
BY: IMRAN

Working with standard modules

- We can use dir command (it is different than the dir command we use in Windows or Linux) to find all the entities in a module.
- But, before using the dir command, the module have been previously imported.

```
import math
for name in dir(math):
print(name, end="\t")
```

- The example code should produce the following output:



```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)
] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh
', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh'
, 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', '
nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'tru
nc']
>>>
```

Selected functions from the math module

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

Selected functions from the math module

```
from math import e, exp, log

print(pow(e, 1) == exp(log(e)))
print(pow(2, 2) == exp(2 * log(2)))
print(log(e, e) == exp(0))
```

Selected functions from the math module

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))
```

Is there real randomness in computers?

- Another module worth mentioning is the one named random.
- It delivers some mechanisms allowing you to operate with **pseudorandom numbers**.
- Note the prefix **pseudo** - the numbers generated by the modules may look random in the sense that you cannot predict their subsequent values, but don't forget that they all are calculated using very refined algorithms.

- The algorithms aren't random - they are deterministic and predictable. Only those physical processes which run completely out of our control (like the intensity of cosmic radiation) may be used as a source of actual random data. Data produced by deterministic computers cannot be random in any way.

Is there real randomness in computers?

- A random number generator takes a value called a **seed**, treats it as an input value, calculates a "random" number based on it (the method depends on a chosen algorithm) and produces a **new seed value**.
- The length of a cycle in which all seed values are unique may be very long, but it isn't infinite - sooner or later the seed values will start repeating, and the generating values will repeat, too. This is normal. It's a feature, not a mistake, or a bug.
- The initial seed value, set during the program start, determines the order in which the generated values will appear.
- The random factor of the process may be **augmented by setting the seed with a number taken from the current time** - this may ensure that each program launch will start from a different seed value (ergo, it will use different random numbers).
- Fortunately, such an initialization is done by Python during module import.

Selected functions from the random module

- The example program below will produce five pseudorandom values - as their values are determined by the current (rather unpredictable) seed value, you can't guess them:

```
from random import random
for i in range(5):
    print(random())
```

Selected functions from the random module

- **The seed function**

- The seed() function is able to directly set the generator's seed. We'll show you two of its variants:

  - seed() - sets the seed with the current time;
  - seed(int_value) - sets the seed with the integer value int_value.

```
from random import random, seed

seed(0)

for i in range(5):
    print(random())
```

Selected functions from the random module

- **The randrange and randint functions**

- If you want integer random values, one of the following functions would fit better:

- randrange(end)
- randrange(beg, end)
- randrange(beg, end, step)
- randint(left, right)

Selected functions from the random module

- **The randrange and randint functions**

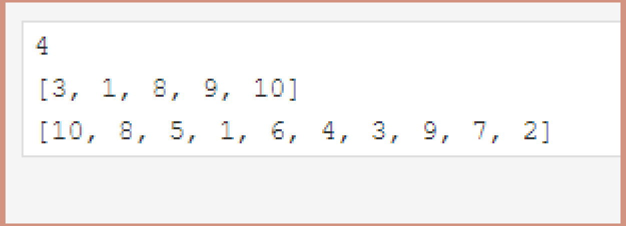
```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1))
```

Module 1: Modules in Python
Part 2: Selected Python Modules

Selected functions from the random module

- `from random import randint`
- `for i in range(10):`
- `print(randint(1, 10), end=',')`
- **The `choice` and `sample` functions**

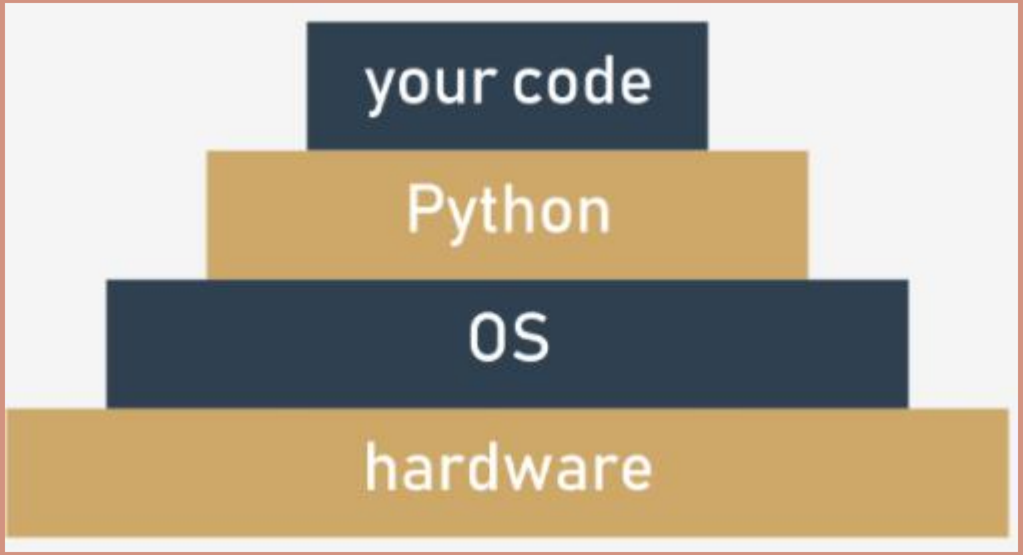
```
from random import choice, sample
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(choice(my_list))
print(sample(my_list, 5))
print(sample(my_list, 10))
```



```
4
[3, 1, 8, 9, 10]
[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]
```

How to know where you are?

- Sometimes, it may be necessary to find out information unrelated to Python. For example, you may need to know the location of your program within the greater environment of the computer.
- Imagine your program's environment as a pyramid consisting of a number of layers or platforms.



How to know where you are?

- The layers are:
  - your (running) code is located at the top of it;
  - Python (more precisely - its runtime environment) lies directly below it;
  - the next layer of the pyramid is filled with the OS (operating system) - Python's environment provides some of its functionalities using the operating system's services; Python, although very powerful, isn't omnipotent - it's forced to use many helpers if it's going to process files or communicate with physical devices;
  - the bottom-most layer is hardware - the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.), and all other machinery needed to make the computer run; the OS knows how to drive it, and uses lots of tricks to conducting all parts in a consistent rhythm.

How to know where you are?

- This means than some of your (or rather your program's) actions have to travel a long way to be successfully performed - imagine that:
  - **your code** wants to create a file, so it invokes one of Python's functions;
  - **Python** accepts the order, rearranges it to meet local OS requirements (it's like putting the stamp "approved" on your request) and sends it down (this may remind you of a chain of command)
  - the **OS** checks if the request is reasonable and valid (e.g., whether the file name conforms to some syntax rules) and tries to create the file; such an operation, seemingly very simple, isn't atomic - it consists of many minor steps taken by...
  - the **hardware**, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

How to know where you are?

- There is a module providing some means to allow you to know where you are and what components work for you. The module is named **platform**.

Selected functions from the platform module

• The platform function

- The platform module lets you access the underlying platform's data, i.e., hardware, operating system, and interpreter version information.

```
from platform import platform

print(platform())
print(platform(1))
print(platform(0, 1))
```

Selected functions from the platform module

• The machine function

- Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code - a function named machine() will tell you that.

```
from platform import machine

print(machine())
```

Selected functions from the platform module

- **The processor function**

  - The processor() function returns a string filled with the real processor name (if possible).

```
from platform import processor

print(processor())
```

Selected functions from the platform module

- **The system function**
  - **A function named system() returns the generic OS name as a string.**

```
from platform import system

print(system())
```

Selected functions from the platform module

- **The version function**

  - The OS version is provided as a string by the version() function.

```
from platform import version

print(version())
```

Selected functions from the platform module

- **The python_implementation and the python_version_tuple functions**

- If you need to know what version of Python is running your code, you can check it using a number of dedicated functions - here are two of them:

  - python_implementation() → returns a string denoting the Python implementation
  - python_version_tuple() → returns a three-element tuple filled with:
    - the major part of Python's version;
    - the minor part;
    - the patch level number.

```
from platform import python_implementation, python_version_tuple
print(python_implementation())
for atr in python_version_tuple():
    print(atr)
```

Python Module Index

- We have only covered the basics of Python modules here. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python.
- Moreover, the Python community all over the world creates and maintains hundreds of additional modules used in very niche applications like genetics, psychology, or even astrology.
- These modules aren't (and won't be) distributed along with Python, or through official channels, which makes the Python universe broader - almost infinite.
- You can read about all standard Python modules here: https://docs.python.org/3/py-modindex.html.