assert

assert is used for debugging purposes.

While programming, sometimes we wish to know the internal state or check if our assumptions are true. assert helps us do this and find bugs more conveniently. assert is followed by a condition.

If the condition is true, nothing happens. But if the condition is false, AssertionError is raised. For example:

```
>> a = 4
>>> assert a < 5
>>> assert a > 5
```

For our better understanding, we can also provide a message to be printed with the `AssertionError`.

```
>>> a = 4
>>> assert a > 5, "The value of a is too small"
```

as

`as` is used to create an alias while importing a module. It means giving a different name (user-defined) to a module while importing it.

As for example, Python has a standard module called `math`. Suppose we want to calculate what cosine pi is using an alias. We can do it as follows using `as`:

```
>>> import math as myAlias
>>>myAlias.cos(myAlias.pi)
-1.0
```

Here we imported the `math` module by giving it the name `myAlias`. Now we can refer to the `math` module with this name. Using this name we calculated cos(pi) and got `-1.0` as the answer.

class

`class` is used to define a new user-defined class in Python.

Class is a collection of related attributes and methods that try to represent a real world situation. This idea of putting data and functions together in a class is central to the concept of object-oriented programming (OOP).

Classes can be defined anywhere in a program. But it is a good practice to define a single class in a module. Following is a sample usage:

```
class ExampleClass:
    def function1(parameters):
        ...
    def function2(parameters):
```

```
    ...
```

def

`def` is used to define a user-defined function.

Function is a block of related statements, which together does some specific task. It helps us organize code into manageable chunks and also to do some repetitive task.

The usage of `def` is shown below:

```
def function_name(parameters):
    ...
```

Learn more about Python functions.

del

`del` is used to delete the reference to an object. Everything is object in Python. We can delete a variable reference using `del`

```
>>> a = b = 5
>>> del a
>>> a
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
NameError: name 'a' is not defined
>>> b
5
```

Here we can see that the reference of the variable `a` was deleted. So, it is no longer defined. But `b` still exists.

`del` is also used to delete items from a list or a dictionary:

```

>>> a = ['x','y','z']
>>> del a[1]
>>> a
['x', 'z']
```

except, raise, try

`except, raise, try` are used with exceptions in Python.

Exceptions are basically errors that suggests something went wrong while executing our

program. `IOError`, `ValueError`, `ZeroDivisionError`, `ImportError`, `NameError`, `Ty`

peError etc. are few examples of exception in Python. try...except blocks are used to catch exceptions in Python.
We can raise an exception explicitly with the raise keyword. Following is an example:

```
def reciprocal(num):
    try:
        r = 1/num
    except:
        print('Exception caught')
        return
    return r

print(reciprocal(10))
print(reciprocal(0))
```

Output

```
0.1
Exception caught
None
```

Here, the function reciprocal() returns the reciprocal of the input number. When we enter 10, we get the normal output of 0.1. But when we input 0, a ZeroDivisionError is raised automatically.
This is caught by our try…except block and we return None. We could have also raised the ZeroDivisionError explicitly by checking the input and handled it elsewhere as follows:

```
if num == 0:
    raise ZeroDivisionError('cannot divide')
```

finally
finally is used with try…except block to close up resources or file streams. Using finally ensures that the block of code inside it gets executed even if there is an unhandled exception. For example:

```
try:
    Try-block
except exception1:
    Exception1-block
except exception2:
```

```
    Exception2-block
else:
    Else-block
finally:
    Finally-block
```

Here if there is an exception in the `Try-block`, it is handled in the `except` or `else` block. But no matter in what order the execution flows, we can rest assured that the `Finally-block`is executed even if there is an error. This is useful in cleaning up the resources.

from, import

`import` keyword is used to import modules into the current namespace. `from…import` is used to import specific attributes or functions into the current namespace. For example:

```
import math
```

will import the `math` module. Now we can use the `cos()` function inside it as `math.cos()`. But if we wanted to import just the `cos()` function, this can done using `from` as

```
from math import cos
```

now we can use the function simply as `cos()`, no need to write `math.cos()`.

in

`in` is used to test if a sequence (list, tuple, string etc.) contains a value. It returns `True` if the value is present, else it returns `False`. For example:

```
>>> a = [1, 2, 3, 4, 5]
>>> 5 in a
True
>>> 10 in a
False
```

The secondary use of `in` is to traverse through a sequence in a `for` loop.

```
for i in 'hello':
    print(i)
```

Output

```
h
e
l
l
```

```
o
```

is

`is` is used in Python for testing object identity. While the `==` operator is used to test if two variables are equal or not, `is` is used to test if the two variables refer to the same object.

It returns `True` if the objects are identical and `False` if not.

```
>>> True is True
True
>>> False is False
True
>>> None is None
True
```

We know that there is only one instance of `True`, `False` and `None` in Python, so they are identical.

```
>>> [] == []
True
>>> [] is []
False
>>> {} == {}
True
>>> {} is {}
False
```

An empty list or dictionary is equal to another empty one. But they are not identical objects as they are located separately in memory. This is because list and dictionary are mutable (value can be changed).

```
>>> '' == ''
True
>>> '' is ''
True
>>> () == ()
True
>>> () is ()
True
```

Unlike list and dictionary, string and tuple are immutable (value cannot be altered once defined). Hence, two equal string or tuple are identical as well. They refer to the same memory location.

lambda

`lambda` is used to create an anonymous function (function with no name). It is an inline function that does not contain a `return` statement. It consists of an expression that is evaluated and returned. For example:

```
a = lambda x: x*2
for i in range(1,6):
    print(a(i))
```

Output

```
2
4
6
8
10
```

Here, we have created an inline function that doubles the value, using the `lambda` statement. We used this to double the values in a list containing 1 to 5.

pass

`pass` is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

Suppose we have a function that is not implemented yet, but we want to implement it in the future. Simply writing,

```
def function(args):
```

in the middle of a program will give us `IndentationError`. Instead of this, we construct a blank body with the `pass` statement.

```
def function(args):
    pass
```

We can do the same thing in an empty `class` as well.

```
class example:
    pass
```

with

`with` statement is used to wrap the execution of a block of code within methods defined by the context manager.

Context manager is a class that implements `__enter__` and `__exit__` methods. Use of `with`statement ensures

that the __exit__ method is called at the end of the nested block. This concept is similar to the use of try…finally block. Here, is an example.

```
with open('example.txt', 'w') as my_file:
    my_file.write('Hello world!')
```

This example writes the text Hello world! to the file example.txt. File objects have __enter__ and __exit__ method defined within them, so they act as their own context manager.

First the __enter__ method is called, then the code within with statement is executed and finally the __exit__ method is called. __exit__ method is called even if there is an error. It basically closes the file stream.

yield

yield is used inside a function like a return statement. But yield returns a generator.

Generator is an iterator that generates one item at a time. A large list of value will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory. For example,

```
>>> g = (2**x for x in range(100))
```

will create a generator g which generates powers of 2 up to the number two raised to the power 99. We can generate the numbers using the next() function as shown below.

```
>>> next(g)
1
>>> next(g)
2
>>> next(g)
4
>>> next(g)
8
>>> next(g)
16
```

And so on… This type of generator is returned by the yield statement from a function. Here is an example.

```
def generator():
    for i in range(6):
        yield i*i
```

```
g = generator()
for i in g:
    print(i)
```

Output

```
0
1
4
9
16
25
```

Here, the function `generator()` returns a generator that generates square of numbers from 0 to 5. This is printed in the `for` loop.