3 Days Training on Python3

Day 2 : Module 8

Muhammad Saufy Rohmad

Module 8 (90 minutes)

Objectives

1. Python Properties
2. Error and Exception Handling

1. Python Properties

- Many object-oriented languages have the explicit concept of encapsulation; that is the ability to hide data within an object and only to provide specific gateways into that data
- These gateways are methods defined to get or set the value of an attribute (often referred to as getters and setters).
- This allows more control over access to the data; for example, it is possible to check that only a positive integer above zero, but below 120, is used for a person’s age etc

1. Python Properties(2)

- Python does not explicitly have the concept of encapsulation;
- instead it relies on two things;
  - a standard convention used to indicate that an attribute should be considered private
  - a concept called a property which allows setters and getters to be defined for an attribute.

1. Python Properties – examine code below

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age
    def get_age(self):
        return self._age
        def set_age(self, new_age):
    if isinstance(new_age,int) & new_age > 0 &
        new_age < 120:
        self._age = new_age
    def get_name(self):
        return self._name
    def __str__(self):
        return 'Person[' + str(self._name) +'] is ' +
        str(self._age)
```

1. Python Properties – examine code below

- Call with

```
person = Person('John', 54)
person.set_age(-1)
print(person)
```

- Then this is ignored, and the person’s age remains as it was, thus the output of this is:

```
Person[John] is 54
```

1.1 Defining Python Properties

```
def get_age(self):
    return self._age
def set_age(self, new_age):
    if isinstance(new_age,int) & new_age > 0 &
    new_age < 120:
    self._age = new_age
    age = property(get_age, set_age, doc="An age property")
def get_name(self):
    return self._name
    name = property(get_name, doc="A name property")
def __str__(self):
    return 'Person[' + str(self._name) +'] is ' +
    str(self._age)
```

1.1 Defining Python Properties(2)

- Notice how we can now write person.age and person.age = 21.
- in both these cases we are accessing the property age that results in the method get_age() and set_age() being executed respectively.
- Thus, the setter is still protecting theupdate to the underlying _age attribute that is actually used to hold the actual value.

1.1 Defining Python Properties(3)

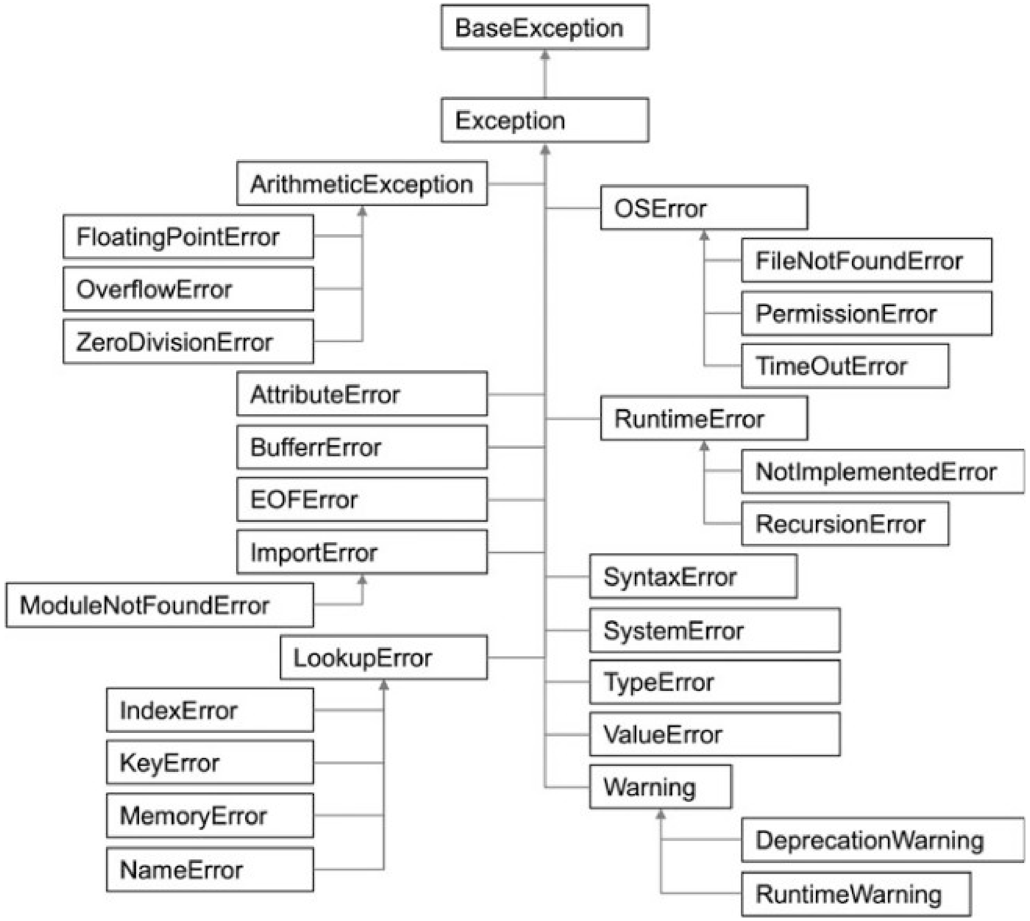
- To overcome this a more concise option has been available since Python 2.4.
- This approach uses what are known as decorators.

```
@property
def age(self):
    """ The docstring for the age property """
    print('In age method')
    return self._age
```

2. Error and Exception Handling

- When something goes wrong in a computer program someone needs to know about it.
- One way of informing other parts of a program (and potentially those running a program) is by generating an error object and propagating that through the code until either something handles the error and sorts thing out or the point at which theprogram is entered is found.
- In Python, everything is a type of object, including integers, strings, booleans and indeed Exceptions and Errors.
- In Python the Exception/Error types are defined in a class hierarchy with the root of this hierarchy being the BaseException type

2. Error and Exception Handling(2)

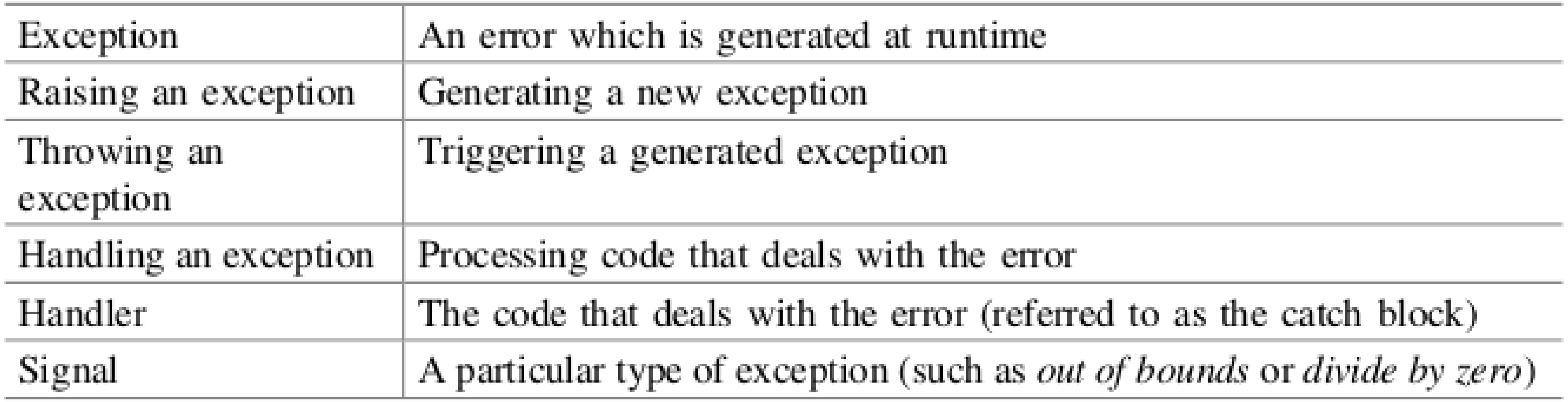


2. Error and Exception Handling(3)

- An exception moves the flow of control from one place to another
- In most situations, this is because a problem occurs which cannot be handled locally but that can be handled in another part of the system.
- The problem is usually some sort of error (such as dividing by zero), although it can be any problem (for example, identifying that the postcode specified with an address does not match)
- The purpose of an exception, therefore, is to handle an error condition when it happens at run time.

2. Error and Exception Handling(4)

- The following table illustrates terminology typically used with exception/error handling in Python.



Exception	An error which is generated at runtime
Raising an exception	Generating a new exception
Throwing an exception	Triggering a generated exception
Handling an exception	Processing code that deals with the error
Handler	The code that deals with the error (referred to as the catch block)
Signal	A particular type of exception (such as *out of bounds* or *divide by zero*)

2.1 Handling an Exception

- You can catch an exception by implementing the try—except construct.

```
def runcalc(x):
    x / 0
```

- If we now call this function, we will get the error trackback in the standard output

```
runcalc(6)
```

2.1 Handling an Exception(2)

- However, we can handle this by wrapping the call to runcalc within a try statement and providing an except clause.
- The syntax for a try statement with an except clause is:

```
try:
    runcalc(6)
except ZeroDivisionError:
    print('oops')
```

2.1 Handling an Exception(3)

- It is possible to gain access to the exception object being caught by the except clause using the as keyword.

```
try:
    runcalc(6)
except ZeroDivisionError as exp:
    print(exp)
    print('oops')
```

2.2 Jumping to Exception Handler

- Below is my_function

```
def my_function(x, y):
    print('my_function in')
    result = x / y
    print('my_function out')
    return result
print('Starting')
```

- And this is normal execution code

```
try:
    print('Before my_function')
    my_function(6, 2)
    print('After my_function')
except ZeroDivisionError as exp:
print('oops')
print('Done')
```

2.2 Jumping to Exception Handler(2)

- When we run this the output is

```
Starting
Before my_function
my_function in
my_function out
After my_function
Done
```

2.2 Jumping to Exception Handler(3)

- If we now change the call to my_function() to pass in 6 and 0 we will raise the ZeroDivisionError.

```
print('Starting')
try:
    print('Before my_function')
    my_function(6, 0)
    print('After my_function')
except ZeroDivisionError as exp:
    print('oops')
print('Done')
```

- Now the output is:

```
Starting
Before my_function
my_function in
oops
Done
```

2.3 Catch Any Exception

- It is also possible to specify an except clause that can be used to catch any type of error or exception, for example:

```
try:
    my_function(6, 0)
except IndexError as e:
    print(e)
except:
    print('Something went wrong')
```

2.4 The else Clause

- The try statement also has an optional else clause. If this is present, then it must come after all except clauses. The else clause is executed if and only if no exceptions were raised.

```
try:
    my_function(6, 2)
except ZeroDivisionError as e:
    print(e)
else:
    print('Everything worked OK')
```

In this case the output is:

```
my_function in
my_function out
Everything worked OK
```

2.5 Raising an Exception

- An error or exception is raised using the keyword raise.

```
def function_bang():
    print('function_bang in')
    raise ValueError('Bang!')
    print('function_bang')
```

- In the above function the second statement in the function body will create a new instance of the ValueError class and then raise it so that it is thrown allowing it to be caught by any exception handlers that have been defined.

```
try:
    function_bang()
except ValueError as ve:
    print(ve)
```

In this case the output is:

```
function_bang in
Bang!
```