

Programming in Python

GAVRILUT DRAGOS

COURSE 4

Exceptions

Exceptions in Python have the following form:

| Python 2.x / 3.x | Python 2.x / 3.x |
|---|--|
| <pre>try: #code except: #code that will be executed in #case of any exception</pre> | <pre>try: x = 5 / 0 except: print("Exception")</pre> |

Output

Exception

Exceptions

Exceptions in Python have the following form:

| Python 2.x / 3.x | Python 2.x / 3.x |
|--|---|
| <pre>try: #code except: #code that will be executed in #case of any exception else: #code that will be executed if #there is no exception</pre> | <pre>try: x = 5 / 1 except: print("Exception") else: print("All ok")</pre> |

Output

All ok

Exceptions

All exceptions in python are derived from **BaseException** class. There are multiple types of exceptions including: **ArithmeticError**, **BufferError**, **AttributeError**, **FloatingPointError**, **IndexError**, **KeyboardInterrupt**, **NotImplementedError**, **OverflowError**, **IndentationError**, and many more.

A list of all the exceptions can be found on:

- <https://docs.python.org/3.5/library/exceptions.html#Exception>
- <https://docs.python.org/2.7/library/exceptions.html#Exception>

A custom (user-defined) exception type can also be created (more on this topic at “Classes”).

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except ArithmeticError:
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")
```

```
Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
Generic exception
All ok
```

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Generic exception")
    except ArithmeticError:
        print("ArithmeticError")
    else:
        print("All ok")
```

```
Test(0)
Test("aaa")
Test(1)
```

Generic exception must be the last one. Code will not compile.

Exceptions

Python also have a finally keyword that can be use to executed something at the end of the try block.

| Python 2.x / 3.x | Python 2.x / 3.x | Output |
|---|--|--|
| <pre>try: #code except: #code for general exception else: #code that will be executed #if there is no exception finally: #code that will be executed #after the try block execution #is completed</pre> | <pre>def Test (y) : try: x = 5 / y except: print ("Error") else: print ("All ok") finally: print ("Final") Test (0) Test (1)</pre> | <pre><u>Test(0):</u> Error Final <u>Test(1):</u> All ok Final</pre> |

Exceptions

Python also have a finally keyword that can be use to executed something at the end of the try block.

Python 2.x / 3.x

```
try:
    #code
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
finally:
    #code that will be executed
    #after the try block execution
    #is completed
```

Python 2.x / 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Error")
    finally:
        print("Final")
    else:
        print("All ok")

Test(0)
Test(1)
```

Finally must be the last statement

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except (Type1, Type2, ... Typen) :
    #code for exception of type
    #1,2,...
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 2.x / 3.x

```
def Test (y) :
    try:
        x = 5 / y
    except (ArithmeticError, TypeError) :
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")
```

```
Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
ArithmeticError
All ok
```

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except Type1 as <var_name>:
    #code for exception of type
    #1.
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 2.x / 3.x

```
try:
    x = 5 / 0
except Exception as e:
    print( str(e) )
```

Output

Python2: integer division or modulo by zero
Python3: division by 0

Exceptions

Exceptions in Python have the following form:

Python 2.x / 3.x

```
try:
    #code
except (Type1, Type2, ... Typen) as <var>:
    #code for exception of type 1,2,... n
```

```
try:
    x = 5 / 0
except (Exception, ArithmeticError, TypeError) as e:
    print( str(e), type(e) )
```

Output

Python2: ('integer division or modulo by zero', <type 'exceptions.ZeroDivisionError'>)

Python3: division by zero <class 'ZeroDivisionError'>

Exceptions

Python also has another keyword (**raise**) that can be use to create or throw an exception:

Python 2.x / 3.x

```
raise ExceptionType (parameters)
raise ExceptionType (parameters) from <exception_var>
```

```
try:
    raise Exception("Testing raise command")
except Exception as e:
    print(e)
```

Output

Testing raise command

Exceptions

Each exception has a list of arguments (parameter *args*)

Python 2.x / 3.x

```
try:
    raise Exception("Param1", 10, "Param3")
except Exception as e:
    params = e.args
    print (len(params))
    print (params[0])
```

Output

```
3
Param1
```

Exceptions

raise keyword can be used without parameters. In this case it indicates that the current exception should be re-raised.

Python 2.x / 3.x

```
try:
    try:
        x = 5 / 0
    except Exception as e:
        print(e)
        raise .....
except Exception as e: ◀.....
    print("Return from raise -> ", e)
```

Output (Python 3.x)

```
division by zero
Return from raise -> division by zero
```

Exceptions

Python 3.x supports chaining exception via **from** keyword.

Python 3.x

```
1  try:
2      x = 5 / 0
3  except Exception as e:
4      raise Exception("Error") from e
```

Output (Python 3.x)

Traceback (most recent call last):

File "a.py", line 2, in <module>

x = 5 / 0

ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

File "a.py", line 4, in <module>

raise Exception("Error") from e

Exception: Error

Exceptions

Python has a special keyword (**assert**) that can be used to raise an exception based on the evaluation of a condition:

Python 2.x / 3.x

```
age = -1
try:
    assert (age>0), "Age should be a positive number"
except Exception as e:
    print (e)
```

Output

Age should be a positive number

Exceptions

pass keyword is usually used if you want to catch an exception but do not want to process it.

Python 2.x / 3.x

```
try:
    x = 10 / 0
except:
    pass
```

Some exceptions (if not handled) can be used for various purposes.

Python 2.x / 3.x

```
print("Test")
raise SystemExit
print("Test2")
```

This exception (**SystemExit**) if not handle will imediately terminate your program

Output

Test

Modules

Modules are python's libraries and extends python functionality. Python has a special keyword (**import**) that can be used to import modules.

Format (Python 2.x/3.x)

```
import module1, [module2, module3, ... modulen]
```

Classes or items from a module can be imported separately using the following syntax.

Format (Python 2.x/3.x)

```
from module import object1, [object2, object3, ... objectn]  
from module import *
```

When importing a module aliases can also be made using “as” keyword

Format (Python 2.x/3.x)

```
import module1 as alias1, [module2 as alias2, ... modulen as aliasn]
```

Modules

Python has a lot of default modules (**os**, **sys**, **re**, **math**, etc).

There is also a keyword (**dir**) that can be used to obtain a list of all the functions and objects that a module exports.

Format (Python 2.x/3.x)

```
import math
print dir(math)
```

Output (Python 3.x)

```
['__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', 'trunc']
```

The list of functions/items from a module may vary from Python 2.x to Python 3.x and from version to version.

Modules

Python distribution modules:

- Python 3.x → <https://docs.python.org/3/py-modindex.html>
- Python 2.x → <https://docs.python.org/2/py-modindex.html>

| Module | Purpose | Module | Purpose |
|-------------|--|------------|---|
| collections | Implementation of different containers | re | Regular expression implementation |
| ctype | Packing and unpacking bytes into c-like structures | random | Random numbers |
| datetime | Date and Time operators | socket | Low-level network interface |
| email | Support for working with emails | subprocess | Processes |
| hashlib | Implementation of different hashes (MD5, SHA, ...) | sys | System specific functions (stdin, stdout, arguments, loaded modules, ...) |
| json | JSON encoder and decoder | traceback | Exception traceback |
| math | Mathematical functions | urllib | Handling URLs / URL requests, etc |
| os | Different functions OS specific (make dir, delete files, rename files, paths, ...) | xml | XML file parser |

Modules - sys

Python documentation page:

- Python 3.x → <https://docs.python.org/3/library/sys.html#sys.modules>
- Python 2.x → <https://docs.python.org/2/library/sys.html#sys.modules>

| object | Purpose |
|--|---|
| sys.argv | A list of all parameters send to the python script |
| sys.platform | Current platform (Windows / Linux / MAC OSX) |
| sys.stdin sys.stdout, sys.stderr | Handlers for default I/O operations |
| sys.path | A list of strings that represent paths from where module will be loaded |
| sys.modules | A dictionary of modules that have been loaded |

Modules - sys

sys.argv provides a list of all parameters that have been send from the command line to a python script. The first parameter is the name/path of the script.

File **'test.py'** (Python 2.x/3.x)

```
import sys
print ("First parameter is", sys.argv[0])
```

Output

```
>>> python.exe C:\test.py
First parameter is C:\test.py
```

Modules - sys

Python 2.x/3.x (File: **sum.py**)

```
import sys
suma = 0
try:
    for val in sys.argv[1:]:
        suma += int(val)
    print("Sum=", suma)
except:
    print("Invalid parameters")
```

Output

```
>>> python.exe C:\sum.py 1 2 3 4
```

```
Sum = 10
```

```
>>> python.exe C:\sum.py 1 2 3 test
```

```
Invalid parameters
```

Modules - os

Python documentation page:

- Python 3.x → <https://docs.python.org/3/library/os.html>
- Python 2.x → <https://docs.python.org/2/library/os.html>

Includes functions for:

- Environment
- Processes (PID, Groups, etc)
- File system (change dir, enumerate files, delete files or directories, etc)
- File descriptor functions
- Terminal informations
- Process management (spawn processes, fork, etc)
- Working with file paths

Modules - os

Listing the contents of a folder (os.listdir → returns a list of child files and folders).

Python 2.x/3.x

```
import os
print (os.listdir("."))
```

Output

```
['$Recycle.Bin', 'Android', 'Documents and Settings', 'Drivers', 'hiberfil.sys', 'Program Files', 'Program Files (x86)', 'ProgramData', 'Python27', 'Python35', 'System Volume Information', 'Users', 'Windows', ...]
```

File and folder operations:

- os.mkdir / os.makedirs → to create folders
- os.chdir → to change current path
- os.rmdir / os.removedirs → to delete a folder
- os.remove / os.unlink → to delete a file
- os.rename / os.rename → rename/move operations

Modules - os

os has a submodule (**path**) that can be use to perform different operations with file/directories paths.

Python 2.x/3.x

```
import os
print (os.path.join ("C:", "Windows", "System32"))
print (os.path.dirname ("C:\\Windows\\abc.txt"))
print (os.path.basename ("C:\\Windows\\abc.txt"))
print (os.path.splitext ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.isdir ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows\\abc.txt"))
```

Output

```
C:\Windows\System32
C:\Windows
abc.txt
["C:\Windows\abc", ".txt"]
False
True
False
False
```

Modules - os

Listing the contents of a folder recursively.

Python 2.x/3.x

```
import os

for (root,directories,files) in os.walk("."):
    for fileName in files:
        full_fileName = os.path.join(root,fileName)
        print (full_fileName)
```

os module can also be used to execute a system command or run an application via **system** function

Python 2.x/3.x

```
import os
os.system("dir *.* /a")
```

Output

```
.\a
.\a.py
.\all.csv
.\run.bat
.\Folder1\version.1.6.0.0.txt
.\Folder1\version.1.6.0.1.txt
.\Folder1\Folder2\version.1.5.0.8.txt
```

Input/Output

Python has 3 implicit ways to work with I/O:

A) IN: via keyboard (with **input** or **raw_input** keywords)

- There are several differences between python 2.x and python 3.x regarding reading from stdin

B) OUT: via **print** keyword

C) IN/OUT: via **open** keyword (to access files)

Input/Output

input keyword performs differently in Python 2.x and Python 3.x:

- In Python 2.x, the content read from the input is evaluated and returned
- In Python 3.x, the content read from the input is considered to be a string and returned

| Format (Python 2.x/3.x) | Python 2.x | Python 3.x |
|---|---|--|
| <pre>input () input (message)</pre> | <pre>>>> Enter: 10 (10, <type 'int'>)</pre> | <pre>>>> Enter: 10 10 <class 'str'></pre> |
| Python 2.x / 3.x | | |
| <pre>x = input("Enter: ") print (x, type (x))</pre> | <pre>>>> Enter: 1+2*3.0 (7.0, <type 'float'>)</pre> | <pre>>>> Enter: 1+2*3.0 1+2*3.0 <class 'str'></pre> |
| | <pre>>>> Enter: "123" ('123', <type 'str'>)</pre> | <pre>>>> Enter: "123" "123" <class 'str'></pre> |
| <ul style="list-style-type: none">○ Use raw_input in Python 2.x to obtain the same effect as in Python 3.x | <pre>>>> Enter: test !!!ERROR!!! (test can not be evaluated)</pre> | <pre>>>> Enter: test test <class 'str'></pre> |

Input/Output

print can be used to print a string in both Python 2 and Python 3. In Python 3 print is a function and supports multiple parameters:

Format (Python 3.x)

```
print (*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Python 3.x

```
>>> print ("test")  
test
```

```
>>> print ("test",10)  
test 10
```

```
>>> print ("test",10,sep="---")  
test---10
```

```
>>> print ("test");print("test2")  
test  
test2
```

```
>>> print ("test",end="***");print("test2")  
test***test2
```

File management

A file can be open in python using the keyword **open**.

Format (Python 3.x)

```
FileObject = open (filePath, mode='r', buffering=-1, encoding=None,  
                    errors=None, newline=None, closefd=True, opener=None)
```

Format (Python 2.x)

```
FileObject = open (filePath, mode='r', buffering=-1)
```

Where mode is a combination of the following:

- "r" – read (default)
- "w" – write
- "x" – exclusive creation (fail if file exists)
- "a" – append
- "b" – binary mode
- "t" – text mode
- "+" – update (read and write)

File management

Python 3 also supports some extra parameters such as:

- encoding → if the file is open in text mode and you need translation from different encodings (UTF, etc)
- error → specify the way conversion errors for different encodings should be processed
- newline → also for text mode, specifies what should be consider a new line. If this value is set to None the character that is specific for the current operating system will be used

Documentation for open function:

- Python 3.x → <https://docs.python.org/3/library/functions.html#open>
- Python 2.x → <https://docs.python.org/2/library/functions.html#open>

File management

A file object has the following methods:

- `f.close` → closes current file
- `f.tell` → returns the current file position
- `f.seek` → sets the current file position
- `f.read` → reads a number of bytes from the file
- `f.write` → write a number of bytes into the file
- `f.readline` → reads a line from the file

Also – the file object is iterable and returns all text lines from a file.

Python 2.x/3.x

```
for line in open("a.py") :  
    print (line.strip())
```

Lines read using this method contain the line-feed terminator. To remove it use `strip` or `rstrip`.

File management

Functional programming can also be used:

Python 2.x/3.x

```
x = [line for line in open("file.txt") if "Gen" in line.strip()]
print (len(x))
```

To read the entire content of the file in a buffer:

Python 2.x/3.x

```
data = open("file.txt", "rb").read()
print (len(data))
print (data[0])
```

read method returns a string in Python 2.x and a buffer in Python 3.x → The output of the previous code will be a character (in Python 2.x) and a number representing the ascii code of that character in Python 3.x

To obtain a string in Python 3.x use “rt” instead of “rb”

File management

To create a file and write content in it:

Python 2.x/3.x

```
open("file.txt", "wt").write("A new file ...")
```

It is a good policy to embed file operation in a try block

Python 2.x/3.x

```
try:  
    f = open("abc.txt")  
    for line in f:  
        print(line.strip())  
    f.close()  
except:  
    print("Unable to open file abc.txt")
```

File management

Once a file is open, the file object handle can be use to retrieve different information regarding that file:

Python 2.x/3.x

```
f = open("a.py", "rb")
print ("File name      : ", f.name)
print ("File open mode : ", f.mode)
print ("Is it closed ? : ", f.closed)
```