python-errors-modules-file-io

October 14, 2020

1 Python for Data Science

Errors, Modules, File IO

2 Errors

Bugs come in three basic flavours:

- *Syntax errors:*
  - Code is not valid Python (easy to fix, except for some whitespace things)
- *Runtime errors:*
  - Syntactically valid code fails, often because variables contain wrong values
- *Semantic errors:*
  - Errors in logic: code executes without a problem, but the result is wrong (difficult to fix)

2.1 Runtime Errors

2.1.1 Trying to access undefined variables

[67]:
```
# Q was never defined
print(Q)
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-67-7ba079dc2063> in <module>
  1 # Q was never defined
----> 2 print(Q)
```

```
NameError: name 'Q' is not defined
```

2.1.2 Trying to execute unsupported operations

[76]:
```
1 + 'abc'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-76-a51a3635a212> in <module>
----> 1 1 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

2.1.3 Trying to access elements in collections that don't exist

[ ]:
```
L = [1, 2, 3]
L[1000]
```

2.1.4 Trying to compute a mathematically ill-defined result

[ ]:
```
2 / 0
```

2.2 Catching Exceptions: `try` and `except`

[77]:
```
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")
```

```
this gets executed first
```

[78]:
```
try:
    print("let's try something:")
    x = 1 / 0 # ZeroDivisionError
except:
```

```
    print("something bad happened!")
```

```
let's try something:
something bad happened!
```

[79]:
```
def safe_divide(a, b):
    """
    A function that does a division and returns a half-sensible
    value even for mathematically ill-defined results
    """
    try:
        return a / b
    except:
        return 1E100
```

[80]:
```
print(safe_divide(1, 2))
print(safe_divide(1, 0))
```

```
0.5
1e+100
```

2.2.1 What about errors that we didn't expect?

[81]:
```
safe_divide (1, '2')
```

[81]:
```
1e+100
```

2.2.2 It's good practice to always catch errors explicitly:

All other errors will be raised as if there were no try/except clause.

[82]:
```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return 1E100
```

[83]:
```
safe_divide(1, '2')
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)
```

```
    <ipython-input-83-cbb3eb91a66d> in <module>
----> 1 safe_divide(1, '2')


    <ipython-input-82-57f0d324952e> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100


TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

2.3 Throwing Errors

- When your code is executed, make sure that it's clear what went wrong in case of errors.
- Throw specific errors built into Python
- Write your own error classes

[84]:
```
raise RuntimeError("my error message")
```

```
␣
↪---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call␣
↪last)

<ipython-input-84-b1834d213d3b> in <module>
----> 1 raise RuntimeError("my error message")


RuntimeError: my error message
```

2.4 Specific Errors

[ ]:
```
def safe_divide(a, b):
    if (not issubclass(type(a), float)) or (not issubclass(type(b), float)):
        raise ValueError("Arguments must be floats")
    try:
        return a / b
    except ZeroDivisionError:
        return 1E100
```

[ ]:
```
safe_divide(1, '2')
```

2.5 Accessing Error Details

[85]:
```
import warnings

def safe_divide(a, b):
    if (not issubclass(type(a), float)) or (not issubclass(type(b), float)):
        raise ValueError("Arguments must be floats")
    try:
        return a / b
    except ZeroDivisionError as err:
        warnings.warn("Caught Error {} with message {}".format(type(err),err) +
                "  - will just return a large number instead")
        return 1E100
```

[86]:
```
safe_divide(1., 0.)
```

```
/Users/felix/anaconda3/envs/pdds_1920/lib/python3.7/site-
packages/ipykernel_launcher.py:10: UserWarning: Caught Error <class
'ZeroDivisionError'> with message float division by zero - will just return a
large number instead
  # Remove the CWD from sys.path while we load stuff.
```

[86]:
```
1e+100
```

3 Loading Modules: the `import` Statement

- Explicit imports (best)
- Explicit imports with alias (ok for long package names)
- Explicit import of module contents
- Implicit imports (to be avoided)

3.1 Creating Modules

- Create a file called [somefilename].py
- In a (i)python shell change dir to that containing dir
- type

```
import [somefilename]
```

Now all classes, functions and variables in the top level namespace are available.

Let's assume we have a file `mymodule.py` in the current working directory with the content:

```
mystring = 'hello world'

def myfunc():
    print(mystring)
```

[87]:
```
import mymodule
mymodule.mystring
```

[87]:
```
'hello world'
```

[88]:
```
mymodule.myfunc()
```

```
hello world
```

3.2 Explicit module import

Explicit import of a module preserves the module's content in a namespace.

[89]:
```
import math
math.cos(math.pi)
```

[89]:
```
-1.0
```

3.3 Explicit module import with aliases

For longer module names, it's not convenient to use the full module name.

[90]:
```
import numpy as np
np.cos(np.pi)
```

[90]:
```
-1.0
```

3.4 Explicit import of module contents

You can import specific elements separately.

[91]:
```
from math import cos, pi
cos(pi)
```

[91]:
```
-1.0
```

3.5 Implicit import of module contents

You can import all elements of a module into the global namespace. Use with caution.

[92]:
```
cos = 0
from math import *
sin(pi) ** 2 + cos(pi) ** 2
```

[92]:
```
1.0
```

4 File IO and Encoding

- Files are opened with `open`
- By default in `'r'` mode, reading text mode, line-by-line

4.1 Reading Text

[93]:
```
path = 'umlauts.txt'
f = open(path)
lines = [x.strip() for x in f]
f.close()
lines
```

[93]:
```
['Eichhörnchen', 'Flußpferd', '', 'Löwe', '', 'Eichelhäher']
```

[94]:
```
# for easier cleanup
with open(path) as f:
    lines = [x.rstrip() for x in f]
lines
```

[94]:
```
['Eichhörnchen', 'Flußpferd', '', 'Löwe', '', 'Eichelhäher']
```

4.2 Detour: Context Managers

Often, like when opening files, you want to make sure that the file handle gets closed in any case.

```
file = open(path, 'w')
try:
    # an error
    1 / 0
finally:
    file.close()
```

Context managers are a convenient shortcut:

```
with open(path, 'w') as opened_file:
    # an error
    1/0
```

4.3 Writing Text

[95]:
```
with open('tmp.txt', 'w') as handle:
    handle.writelines(x for x in open(path) if len(x) > 1)
[x.rstrip() for x in open('tmp.txt')]
```

[95]:
```
['Eichhörnchen', 'Flußpferd', 'Löwe', 'Eichelhäher']
```

4.4 Reading Bytes

[96]:
```
# remember 't' was for text reading/writing
with open(path, 'rt') as f:
    # just the first 6 characters
    chars = f.read(6)
chars
```

[96]:
```
'Eichhö'
```

[97]:
```
# now we read the file content as bytes
with open(path, 'rb') as f:
    # just the first 6 bytes
    data = f.read(6)
```

[98]:
```
# byte representation
data.decode('utf8')
```

```
␣
↪---------------------------------------------------------------------------

UnicodeDecodeError                        Traceback (most recent call␣
↪last)

<ipython-input-98-9981ac9de387> in <module>
  1 # byte representation
----> 2 data.decode('utf8')


UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 5:␣
↪unexpected end of data
```

[99]:
```
# decoding error, utf-8 has variable length character encodings
data[:4].decode('utf8')
```

[99]:
```
'Eich'
```