

Errors and Exceptions

Week 6, Lecture 1

source: Errors and Exceptions – Python v2.7 documentation
<http://docs.python.org/tutorial/errors.html>

Outline

- Syntax errors
- Exceptions
- `try` and `except`
- `raise`
- named parameters and defaults

Two types of errors

- Syntax errors – also called parsing errors
 - Python can't run it because you typed something that violates the rules of constructing expressions in python
- Exceptions -

Syntax errors

- Python “parses” what you type, token by token
- If there's an error it has to stop!
- Python repeats the offending line and displays ^ under the token *after* python detected the error.

```
>>> while True print 'Hello world'  
File "<stdin>", line 1, in ?  
    while True print 'Hello world'  
                ^  
SyntaxError: invalid syntax
```

What's wrong here?

Syntax errors

- `<stdin>` is the stream of data coming from your typing!
- If this were a file, python will tell you the line number in the file.

```
>>> while True print 'Hello world'  
File "<stdin>", line 1, in ?  
    while True print 'Hello world'  
                ^  
SyntaxError: invalid syntax
```

Exceptions

- These are sometimes called run-time errors
 - They occur when the program is running.
- The program syntax is correct – python can execute the commands – but the program contains some logical errors that prevent it from completing

Exceptions – syntax is fine, but your logic is wrong!

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

Python tells you what kind of error you made – in this case, you tried to use a variable that had not yet been defined. This is a NameError.

Traceback: Let's you "trace" back from the place in your code back through the method that has your error, back to the command that started the whole thing.

Traceback

example.py – first calls second calls third, which attempts divide by zero (illegal)

```
>>> import example as e
>>> e.first()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 2, in first
    return second()
  File "example.py", line 5, in second
    return third()
  File "example.py", line 8, in third
    return 1/0
ZeroDivisionError: integer division or modulo by zero
>>> 
```

```
def first():
    return second()
def second():
    return third()
def third():
    return 1/0
```

Exceptions come in different types

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Use `try..except` to handle exceptions in your programs

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
... 
```

- python tries to execute the “try” clause code
- if it encounters a `ValueError`, it halts execution of the code in the try clause and then runs the code in the except clause

How do you know in advance what type of errors?

- Use the interpreter to test snippets of code to find out the classes of Exceptions your code may encounter
 - See <http://docs.python.org/library/exceptions.html#builtin-exceptions>
- Note that if you enter ^C, it generates a KeyboardInterrupt

`except` can catch more than one type of Exception

```
>>> import example as e
>>> e.demo(1,2)
0.5
>>> e.demo('1','2')
0.5
>>> e.demo('one','2')
Warning!
>>> e.demo('1','0')
Warning!
>>> 
```

example.py

```
def demo(n,d):
    try:
        return float(n)/float(d)
    except (ZeroDivisionError,ValueError):
        sys.stderr.write("Warning!\n")
        return None
```

- This let's you catch and handle more than one kind of mistake.

More useful is to handle each type of error in its own clause

example.py

```
>>> e.demo2('1',0)
ZeroDivision
<type 'exceptions.ZeroDivisionError'>
('float division',)
float division
>>> e.demo2('1','b')
ValueError
<type 'exceptions.ValueError'>
('invalid literal for float(): b',)
invalid literal for float(): b
```

```
def demo2(n,d):
    try:
        return float(n)/float(d)
    except ZeroDivisionError as z:
        print "ZeroDivision"
        print type(z)
        print z.args
        print z
        return None
    except ValueError as v:
        print "ValueError"
        print type(v)
        print v.args
        print v
        return None
```

- Exceptions have values associated with them, use “as” to create a new variable associated with the Exception object

Exceptions contain data that reveals what went wrong

```
>>> e.demo2('1',0)
ZeroDivision
<type 'exceptions.ZeroDivisionError'>
('float division',)
float division
>>> e.demo2('1','b')
ValueError
<type 'exceptions.ValueError'>
('invalid literal for float(): b',)
invalid literal for float(): b
```

type

arguments – a tuple, the message is always the first item

For printing, Exceptions implement `__str__()` method

- When you print an Exception, python will invoke its `__str__` method, which returns the first argument (a string)

Your code can also raise new Exceptions, using `raise`

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)      # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst          # __str__ allows args to printed directly
...     x, y = inst         # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

- You can also access an exception's arguments using assignment
- Python interpreter uses the Exception object's `__getitem__` accessor to support this

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StandardError
+-- BufferError
+-- ArithmeticError
+-- FloatingPointError
+-- OverflowError
+-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- EnvironmentError
+-- IOError
+-- OSError
+-- WindowsError (Windows)
+-- VMSError (VMS)
+-- EOFError
+-- ImportError
+-- LookupError
+-- IndexError
+-- KeyError
+-- MemoryError
+-- NameError
+-- UnboundLocalError
+-- ReferenceError
+-- RuntimeError
+-- NotImplementedError
+-- SyntaxError
+-- IndentationError
+-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
+-- UnicodeError
+-- UnicodeDecodeError
+-- UnicodeEncodeError
+-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
```

Exception hierarchy

See:

<http://docs.python.org/library/exceptions.html#builtin-exceptions>

For built-in exceptions and their meanings.

Commonly occurring Exceptions with examples

- `AttributeError`
 - happens when you try to access an instance variable that doesn't exist
- `IOError`
 - when you try to open a file that doesn't exist
- `ImportError`
 - when you try to import a module and python can't find it
- `NameError`
 - when you try to use a variable (name) that isn't defined

ValueError versus TypeError

exception **TypeError**

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception **ValueError**

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

TypeError or ValueError?

- Method expects a positive integer, but gets a negative integer instead.

```
def test(v):  
    if not v > 0:  
        raise [FILL IN]  
    else:  
        return v
```

TypeError or ValueError?

- Method expects an integer, but gets a string instead.

```
def test(v):  
    if not type(v) == type(1):  
        raise [FILL IN]  
    else:  
        return v
```

Tip

- Make a "toy" instance of the exception you're considering using, try it out!

```
def getValueError():  
    try:  
        int('s')  
    except ValueError as v:  
        return v
```

Named parameters

- Also called keyword arguments
- Use them to set defaults, permit order to change in calling routines
 - concatenate strings in a list using sep, and print to fh

```
def writeStrings(fh=None,
                sep=' ',
                strs=None)
    fh.write(sep.join(strs)+'\n')
```

```
>>> e.writeStrings(sys.stderr,sep='\t',strs=['a','b'])
a  b
```

Rules for named parameters

- You can combine named and non-named parameters, but named always come first

```
>>> def test(a='a',b,c='c'):
...     print "hello"
...
File "<stdin>", line 1
SyntaxError: non-default argument
follows default argument
>>>
```

Rules for named parameters

- You can specify defaults, or None

```
>>> def test(a=None,b=None,c='c'):  
...     print 'a'+b+c  
...  
>>> test(a='A',b='b')  
abc
```

Named arguments