California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Tutorial

Section 8

Errors and Exceptions

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

8.1. Syntax Errors

- Syntax (parsing) errors are the most common kind of complaint you get while you are still learning Python
- The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected
- The error is caused by (or at least detected at) the token preceding the arrow
- In the example, the error is detected at the function `print()`, since a colon '`:`' is missing before it
- File name and line number are printed so you know where to look in case the input came from a script

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

8.2. Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it
- Errors detected during execution are called exceptions and are not unconditionally fatal
- The last line of the error message indicates what happened
- Exceptions come in different types, and the type is printed as part of the message
  - *ZeroDivisionError*
  - *NameError*
  - *TypeError*
- The string printed as the exception type is the name of the built-in exception that occurred
- This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention)
- Standard exception names are built-in identifiers (not reserved keywords)
- The rest of the line provides detail based on the type of exception and what caused it
- The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

8.3. Handling Exceptions

- It is possible to write programs that handle selected exceptions
- The `try` statement works as follows
  - First, the `try` clause (the statement(s) between the `try` and `except` keywords) is executed
  - If no exception occurs, the `except` clause is skipped and execution of the `try` statement is finished
  - If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the `except` clause is executed, and then execution continues after the `try` statement
  - If an exception occurs which does not match the exception named in the `except` clause, it is passed on to outer `try` statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above

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

- A `try` statement may have more than one except clause, to specify handlers for different exceptions
- An `except` clause may name multiple exceptions as a parenthesized tuple

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

8.3. Handling Exceptions (cont.)

- A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an `except` clause listing a derived class is not compatible with a base class)
- For example, the following code will print B, C, D in that order
- Note that if the `except` clauses were reversed (with except B first), it would have printed B, B, B — the first matching `except` clause is triggered

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

- The last `except` clause may omit the exception *name(s)*, to serve as a wildcard
- Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-`raise` the exception (allowing a caller to handle the exception as well)

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

8.3. Handling Exceptions (cont.)

- The `try` … `except` statement has an optional `else` clause, which, when present, must follow all `except` clauses
- It is useful for code that must be executed if the `try` clause does not raise an exception
- The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` … `except` statement
- When an exception occurs, it may have an associated value, also known as the exception's argument
- The presence and type of the argument depend on the exception type

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

8.3. Handling Exceptions (cont.)

- The `except` clause may specify a *variable* after the exception name
- The variable is bound to an exception instance with the arguments stored in instance `.args`
- For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`
- One may also instantiate an exception first before raising it and add any attributes to it as desired
- If an exception has arguments, they are printed as the last part '*detail*' of the message for unhandled exceptions

```
>>> 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 be printed
directly,
...                          # but may be overridden in exception
subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

8.3. Handling Exceptions (cont.)

- Exception handlers don't just handle exceptions if they occur immediately in the `try` clause
- But also if they occur inside functions that are called (even indirectly) in the `try` clause

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4. Raising Exceptions

- The `raise` statement allows the programmer to force a specified exception to occur
- The sole argument to `raise` indicates the exception to be raised
- This must be either an exception instance or an exception class (a class that derives from `Exception`)
- If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments
- If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

```
raise ValueError  # shorthand for 'raise ValueError()'
```

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. Exception Chaining

- The `raise` statement allows an optional `from` which enables chaining exceptions
- This can be useful when you are transforming exceptions
- Exception chaining happens automatically when an exception is raised inside an `except` or `finally` section

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

```
>>> def func():
...     raise IOError
...
>>> try:
...     func()
... except IOError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
OSError

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

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

- Exception chaining can be disabled by using `from None` idiom
- For more information about chaining mechanics, see *Built-in Exceptions*

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

8.6. User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class
- Exceptions should typically be derived from the `Exception` class, either directly or indirectly
- Exception classes are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception
- When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions
- Most exceptions are defined with names that end in "`Error`", similar to the naming of the standard exceptions
- Many standard modules define their own exceptions to report errors that may occur in functions they define

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's
not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is
not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

8.7. Defining Clean-up Actions

- The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances
- If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes
- The `finally` clause runs whether or not the `try` statement produces an exception
- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue` or `return` statement, the `finally` clause will execute just prior to the `break`, `continue` or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

8.7. Defining Clean-up Actions (cont.)

- The `finally` clause is executed in any event
- The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed
- In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

8.8. Predefined Clean-up Actions

- Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed
- The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing

```
for line in open("myfile.txt"):
    print(line, end="")
```

- This is not an issue in simple scripts, but can be a problem for larger applications
- The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

- After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines
- Objects which, like files, provide predefined clean-up actions will indicate this in their documentation