Programming Principles in Python (CSCI 503)

Exceptions

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Object-Based Programming

• With Python's libraries, you often don't need to write your own classes. Just
  - Know what libraries are available
  - Know what classes are available
  - Make objects of existing classes
  - Call their methods

• With inheritance and overriding and polymorphism, we have true object-oriented programming (OOP)
Named Tuples & SimpleNamespace

- Named tuples add the ability to use dot-notation

```python
from collections import namedtuple
Car = namedtuple('Car', ['make', 'model', 'year', 'color'])
car1 = Car(make='Toyota', model='Camry', year=2000, color="red")
```

- SimpleNamespace does allow mutation:

```python
from types import SimpleNamespace
car2 = SimpleNamespace(make='Toyota', model='Camry', year=2000, color="red")
```

- Access via dot-notation:
  - `car1.make` # "Toyota"
  - `car2.year` # 2000
Typing

• Dynamic Typing: variable's type can change (what Python does)
• Static Typing: compiler enforces types, variable types generally don't change
• Duck Typing: check method/attribute existence, not type
• Python is a dynamically-typed language (and plans to remain so)
• …but it has recently added more support for type hinting/annotations that allow static type checking
• Type annotations change nothing at runtime!
Type Annotations

• def area(width : float, height : float) -> float:
  return width * height

• colon (:) after parameter names, followed by type

• arrow (->) after function signature, followed by type (then final colon)

• area("abc", 3) # runs, returns "abcabcabc"

• These won't prevent you from running this function with the wrong arguments or returning a value that doesn't satisfy the type annotation

• Can use mypy to do static type checking based on annotations
When to use typing

- **Pros:** Good for documentation, Improve IDEs and linters, Build and maintain cleaner architecture
- **Cons:** Takes time and effort!, Requires modern Python, Some penalty for typing imports (can be alleviated)
- **No when learning Python**
- **No for short scripts, snippets in notebooks**
- **Yes for libraries, especially those used by others**
- **Yes for larger projects to better understand flow of code**
Data Classes

- from dataclasses import dataclass
  @dataclass
  class Rectangle:
      width: float
      height: float

- Rectangle(34, 21) # just works!

- Need type annotations, but can use typing.Any

- Does a lot of boilerplate tasks
  - Creates basic constructor (__init__)
  - Creates __repr__ method
  - Creates comparison dunder methods (==, !=, <, >, <=, >=)
Assignment 6

• Object-oriented Programming
• Track University Enrollment
• Methods for checking conflicts (e.g. disallow student to have overlapping courses, take too many credits)
• Methods for changing course time (check the new time works for everyone)
• Sample code is meant to be run in different cells!
• Due Friday
Dealing with Errors

- Can explicitly check for errors at each step
  - Check for division by zero
  - Check for invalid parameter value (e.g. string instead of int)
- Sometimes all of this gets in the way and can't be addressed succinctly
  - Too many potential errors to check
  - Cannot handle groups of the same type of errors together
- Allow programmer to determine when and how to handle issues
  - Allow things to go wrong and handle them instead
  - Allow errors to be propagated and addressed once
Advantages of Exceptions

• Separate error-handling code from "regular" code
• Allows propagation of errors up the call stack
• Errors can be grouped and differentiated
Try-Except

• The `try` statement has the following form:
  
  ```python
  try:
      <body>
  except <ErrorType>*:
      <handler>
  ```

• When Python encounters a `try` statement, it attempts to execute the statements inside the body.

• If there is no error, control passes to the next statement after the `try...except` (unless `else` or `finally` clauses)

• Note: `except` not catch
Try-Except

• If an error occurs while executing the body, Python looks for an except clause with a matching error type. If one is found, the handler code is executed.

```python
try:
c = a / b
except ZeroDivisionError:
c = 0
```

• Without the except clause (or one that doesn't match), the code **crashes**
Exception Hierarchy

• Python's `BaseException` class is the base class for all exceptions

• Four primary subclasses:
  - `SystemExit`: just terminates program execution
  - `KeyboardInterrupt`: occurs when user types Ctrl+C or selects Interrupt Kernel in Jupyter
  - `GeneratorExit`: generator done producing values
  - `Exception`: most exceptions subclass from this!
    • `ZeroDivisionError`, `NameError`, `ValueError`, `IndexError`

• Most exception handling is done for these exceptions
Exception Hierarchy

- Except clauses match when error is an instance of specified exception class
- Remember `isinstance` matches objects of subclasses!
- try:
  
  ```python
  c = a / b
  except Exception:
      c = 0
  ```
  
- Can also have a **bare** except clause (matches any exception!)
- try:
  
  ```python
  c, d = a / b
  except:
      c, d = 0, 0
  ```
  
- ...but DON'T do this!
Exception Granularity

• If you catch any exception using a base class near the top of the hierarchy, you may be **masking** code errors

```python
try:
c, d = a / b
except Exception:
c, d = 0, 0
```

• Remember **Exception** catches any exception is an instance of Exception

• **Catches** `TypeError: cannot unpack non-iterable float object`

• Better to have more **granular** (specific) exceptions!

• We don't want to catch the `TypeError` because this is a **programming error** not a runtime error
Exception Locality

• Generally, want try statement to be specific to a part of the code

```python
try:
    with open('missing-file.dat') as f:
        lines = f.readlines()
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print("An error occurred processing files.")
```

• We don't know whether reading failed or writing failed
• Maybe that is ok, but having multiple try-except clauses might help
Exception Locality

- try:
  
  ```python
  fname = 'missing-file.dat'
  with open(fname) as f:
      lines = f.readlines()
  except OSError:
    print(f"An error occurred reading {fname}")
  try:
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
  except OSError:
    print(f"An error occurred writing {out_fname}")
  ```
Multiple Except Clauses

• May also be able to address with **multiple** except clauses:

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except PermissionError:
    print(f"Cannot write to {out_fname}\")
```

• However, other OSError problems (disk full, etc.) won't be caught
Multiple Except Clauses

• Function like an if/elif sequence
• Checked in order so put more granular exceptions earlier!

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except OSError:
    print("An error occurred processing files")
```
Multiple Except Clauses

- Function like an if/elif sequence
- Checked in order so put more granular exceptions earlier!

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print("An error occurred processing files")
except FileNotFoundError:
    print(f"File {fname} does not exist")
```
Multiple Except Clauses

- Function like an if/elif sequence
- Checked in order so put more granular exceptions **earlier**!

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print("An error occurred processing files")
except FileNotFoundError:
    print(f"File {fname} does not exist")
```
Bare Except

- The bare except clause acts as a catch-all (elif any other exception)

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except OSError:
    print("An error occurred processing files")
except:
    print("Any other error goes here")
```
Handling Multiple Exceptions at Once

- Can process multiple exceptions with one clause, use `tuple` of classes
- Allows some specificity but without repeating

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except (FileNotFoundError, PermissionError):
    print("An error occurred processing files")
```
Exception Objects

• Exceptions themselves are a type of object.
• If you follow the error type with an identifier in an except clause, Python will assign that identifier the actual exception object.
• Sometimes exceptions encode information that is useful for handling

```python
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError as e:
    print(e.errno, e.filename, e)
```
Else Clause

- Code that executes if no exception occurs
- \( b = 3 \)
  \( a = 2 \)

```python
try:
    c = b / a
except ZeroDivisionError:
    print("Division failed")
    c = 0
else:
    print("Division successful:", c)
```
Finally

- Code that always runs, **regardless** of whether there is an exception

```python
b = 3
a = 0
try:
    c = b / a
except ZeroDivisionError:
    print("Division failed")
    c = 0
finally:
    print("This always runs")
```
Finally

• Code that always runs, **regardless** of whether there is an exception
• ...even if the exception isn't handled!

```python
b = 3
a = 0
try:
    c = b / a
finally:
    print("This always runs, even if we crash")
```

• Remember that context managers (e.g. for files) have built-in cleanup clauses
Nesting

- You can nest try-except clauses inside of except clauses, too.
- Example: perhaps a file load could fail so you want to try an alternative location but want to know if that fails, too.
- Can even do this in a `finally` clause:

```python
try:
    c = b / a
finally:
    try:
        print("This always runs", 3/0)
    except ZeroDivisionError:
        print("It is silly to only catch this exception")
```
Raising Exceptions

• Create an exception and raise it using the `raise` keyword
• Pass a string that provides some detail
• Example: `raise Exception("This did not work correctly")`
• Try to find an exception class:
  - ValueError: if an argument doesn't fit the functions expectations
  - NotImplementedError: if a method isn't implemented (e.g. abstract cls)
• Be specific in the error message, state actual values
• Can also subclass from existing exception class, but check if existing exception works first
• Some packages create their own base exception class (`RequestException`)
Re-raising and Raising From

• Sometimes, we want to detect an exception but also pass it along
  try:
      c = b / a
  except ZeroDivisionError:
      print("Division failed")
      raise

• Raising from allows exception to show specific chain of issues
  try:
      c = b / a
  except ZeroDivisionError as e:
      print("Division failed")
      raise ValueError("a cannot be zero") from e

• Usually unnecessary because Python does the right thing here (shows chain)
Making Sense of Exceptions

- When code (e.g. a cell) crashes, read the traceback:

  ZeroDivisionError: division by zero
Making Sense of Exceptions

• Start at the bottom: last line is the exception message
• Nesting goes outside-in: innermost scope is last, outermost scope is first
• Arrows point to the line of code that caused errors at each scope
• Surrounding lines give context
Making Sense of Exceptions

- Sometimes, exception handling can mask actual issue!

```python
def process(a, b):
    ...
    for i in range(4):
        try:
            process(3, i)
        except ZeroDivisionError:
            raise Exception(f"Cannot process i={i}" ) from None
```

- Exception Traceback (most recent call last)
  <ipython-input-60-6d0289010945> in <module>
    7         process(3, i)
    8     except ZeroDivisionError:
       ---> 9         raise Exception(f"Cannot process i={i}" ) from None
Exception: Cannot process i=3

- Usually, Python includes inner exception (from None stops the chain)
Making Sense of Exceptions

• Probably the **worst** thing is to **ignore** all exceptions:

```python
def process(a, b):
    ...
    result = []
    for i in range(6):
        try:
            result.append(process(3, i))
        except:
            pass
```

• This may seem like the easy way out, don't have to worry about errors, but can mask major issues in the code!

• Be specific (granularity), try to handle cases when something goes wrong, crash **gracefully** if it is an unexpected error