Programming Principles in Python (CSCI 503/490)

Exceptions

Dr. David Koop
Quiz
Question 1

• Which of the following variables is intended to be private?
  (a) _attr_
  (b) _attr
  (c) private:attr
  (d) __attr
Question 2

• Which syntax is used to define a property using a decorator?
  (a) @property
  (b) $property
  (c) #property
  (d) &property
Question 3

• Which of the following is true?
  (a) Python defines instance variables outside of methods
  (b) Python uses the base method to access base class definitions
  (c) Python allows multiple inheritance
  (d) Python does not allow class methods
Question 4

• Which class do all Python classes (indirectly) inherit from?
  (a) cls
  (b) object
  (c) abc
  (d) None of the above
Question 5

• Given a class `Vehicle`, which is a valid constructor signature?
  (a) `def Vehicle(self, make, model)`
  (b) `def Vehicle(this, make, model)`
  (c) `def __init__(self, make, model)`
  (d) `def constructor(this, make, model)`
Duck Typing

• "If it looks like a duck and quacks like a duck, it must be a duck."
• Python "does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used"
• class Rectangle:
  def area(self):
    ...
• class Circle:
  def area(self):
    ...
• It doesn't matter that they don't have a common base class as long as they respond to the methods/attributes we expect: shape.area()
Multiple Inheritance

- Can have a class inherit from two different superclasses
- HybridCar inherits from Car and Hybrid
- Python allows this!
  - class HybridCar(Car, Hybrid): ...
- Problem: how is super() is defined?
  - Diamond Problem
  - Python use the method resolution order (MRO) to determine order of calls
Method Resolution Order

• The order in which Python checks classes for a method
• `mro()` is a **class** method
• `Square.mro()` # `['__main__.Square, __main__.Rectangle, object']`

• Order of base classes matters:
  - class HybridCar(Car, Hybrid):
    pass
    HybridCar.mro() # `['__main__.HybridCar, __main__.Car, __main__.Hybrid, __main__.Vehicle, object']`
  - class HybridCar(Hybrid, Car):
    pass
    HybridCar.mro() # `['__main__.HybridCar, __main__.Hybrid, __main__.Car, __main__.Vehicle, object']"
Assignment 5

• Due Friday
• Writing a Python Package and Command-Line Tools
• Same Pokémon data
• Analysis and Comparison
• Create package and command-line tool
Named Tuples

- Tuples are immutable, but cannot refer to with attribute names, only indexing
- 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")
```

- Can use kwargs or positional or mix
- `car2 = Car('Ford', 'F150', 2018, 'gray')`
- Access via dot-notation:
  - `car1.make # "Toyota"
  - `car2.year # 2018`
SimpleNamespace

• Named tuples do not allow mutation
• SimpleNamespace does allow mutation:

• from types import SimpleNamespace
car3 = SimpleNamespace(make='Toyota', model='Camry',
year=2000, color="red")
• car3.num_doors = 4 # would fail for namedtuple
• Doesn't enforce any structure, though
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

• Extensions for collections allows inner types to be specified:
  - from typing import List
    names : List[str] = ['Alice', 'Bob']

• Any and Optional, too
mypy

• A static type checker for Python that uses the type annotations to check whether types work out
• $ mypy <script.py>
  - Writes type errors tagged by the line of code that introduced them
  - Can also reveal the types of variables at various parts of the program
• There is an extension for Jupyter (mypy_ipython), but it basically works by converting all cells to a script and then running mypy
  - Cells not tagged in error messages
  - Re-running cells introduces multiple copies of error
  - Deleting cells doesn't remove errors
Type Checking in Development Environments

• PyCharm can also use the type hints to do static type checking to alert programmers to potential issues
• Microsoft VS Code Integration using Pyright
Type Checking Pros & Cons

• 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)
When to use typing

- 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!

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

- Requires type annotations, but just like other type annotations, they are not checked at runtime!
- Rectangle("abc", "def") # no error!
- Use mypy to check typing
- If typing is not important, use typing.Any for types
- from typing import Any
  from dataclasses import dataclass
  @dataclass
class Rectangle:
    width: Any
    height: Any
Data Classes

• Can add methods as normal
• from dataclasses import dataclass
  @dataclass
class Rectangle:
    width: float
    height: float

    def area(self):
        return self.width * self.height

• Supports factory methods for more complicated inits
• __post_init__ method for extra processing after __init__
Exceptions
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.

- 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!

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

• Can also have a `bare` except clause (matches any exception!)

```python
• try:
    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
• try:
  
  ```python
  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:
    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!
• try:
  
  ```python
  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!
- try:
  ```python
  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!
- try:
  ```python
  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)
- try:
  ```python
  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
- `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")`