Programming Principles in Python (CSCI 503/490)

Exceptions

Dr. David Koop





D. Koop, CSCI 503/490, Fall 2022

Quiz









- Which decorator is used to define a class method in Python? (a) @classmethod (b) @cls (C) %cls
 - (d) %class









• Which of the following is **false**? (a) dict is a class (b) Python does not allow multiple inheritance (c) Python uses the super method to access base class definitions (d) Python classes may have class variables





- Which class do all Python classes (indirectly) inherit from? (a) object
 - (b) base
 - (C) class
 - (d) None of the above









• Given a class Vehicle, which is a valid constructor signature? (a) def Vehicle(self, make, model) (b) def constructor(this, make, model) (C) def Vehicle(this, make, model) (d) def init (self, make, model)









• Which of the following attributes is intended to be **protected**? (a) attr (b) protected:attr (C) _____attr___ (d) attr

D. Koop, CSCI 503/490, Fall 2022





7

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):
- respond to the methods/attributes we expect: shape.area()

It doesn't matter that they don't have a common base class as long as they









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





9

Method Resolution Order

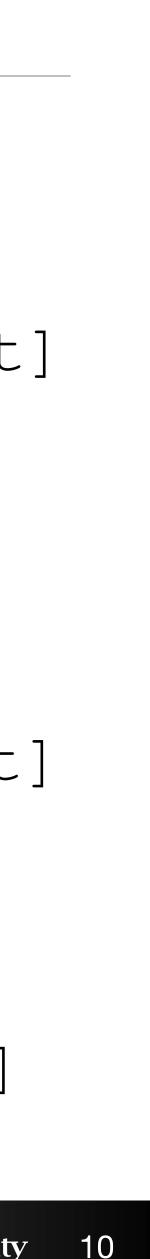
- The order in which Python checks classes for a method
- mro() is a **class** method
- Order of base classes matters:
 - class HybridCar(Car, Hybrid): pass HybridCar.mro() # [main .HybridCar, main .Car,
 - class HybridCar(Hybrid, Car): pass main

• Square.mro() # [main .Square, main .Rectangle, object]

main .Hybrid, main .Vehicle, object]

HybridCar.mro() # [main .HybridCar, main .Hybrid, .Car, main .Vehicle, object]





<u>Assignment 5</u>

- Due Monday
- Writing a Python Package and Command-Line Tools
- Same port entries data
- Structure as dictionaries
- Find by name and state
- Compare measure values
- [CSCI 503] Filter measures





Mixins

- Sometimes, we just want to add a classes
- For example: print_as_dict()
- A mixin class allows us to specify o second
- Caution: Python searches from left right with mixing

• Sometimes, we just want to add a particular method to a bunch of different

• A mixin class allows us to specify one or more methods and add it as the

• Caution: Python searches from left to right so a base class should be at the





Object-Based Programming

- - 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 objectoriented programming (OOP)

• With Python's libraries, you often don't need to write your own classes. Just







13

Named Tuples

- Tuples are immutable, but cannot refer to with attribute names, only indexing Named tuples add the ability to use dot-notation
- 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:
 - carl.make # "Toyota"
 - car2.year # 2018



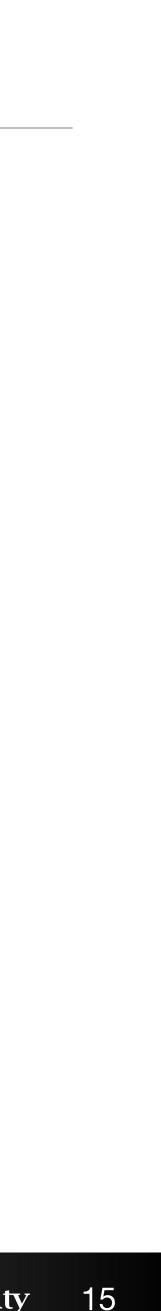




SimpleNamespace

- 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





lyping

- 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





ΓΙΥΡΥ

- whether types work out
- \$ mypy <script.py>
 - converting all cells to a script and then running mypy
- 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
 - Cells not tagged in error messages
 - Re-running cells introduces multiple copies of error
 - Deleting cells doesn't remove errors

• A static type checker for Python that uses the type annotations to check





Type Checking in Development Environments

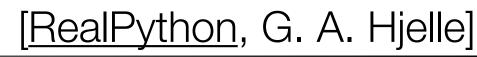
- PyCharm can also use the type hints to do static type checking to alert programmers to potential issues
- Microsoft VS Code Integration using <u>Pyright</u>





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

- 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 Qdataclass

class Rectangle: width: Any

height: Any

D. Koop, CSCI 503/490, Fall 2022

• Requires type annotations, but just like other type annotations, they **are not**







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







D. Koop, CSCI 503/490, Fall 2022

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: 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

- try:

c = a / bexcept ZeroDivisionError: C = 0

D. Koop, CSCI 503/490, Fall 2022

• 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.

• 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 Crl+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

- Remember isinstance matches objects of subclasses!
- try: c = a / bexcept Exception: C = 0
- Can also have a bare except clause (matches any exception!)
- try: c, d = a / bexcept: c, d = 0, 0
- ...but DON'T do this!

D. Koop, CSCI 503/490, Fall 2022

Except clauses match when error is an instance of specified exception class







Exception Granularity

- you may be **masking** code errors
- try: c, d = a / bexcept 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

D. Koop, CSCI 503/490, Fall 2022

• If you catch any exception using a base class near the top of the hierarchy,









Exception Locality

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

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}")







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

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







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

• 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")







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

• 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")





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

• try:

fname = 'missing-file.dat'

with open(fname) as f:

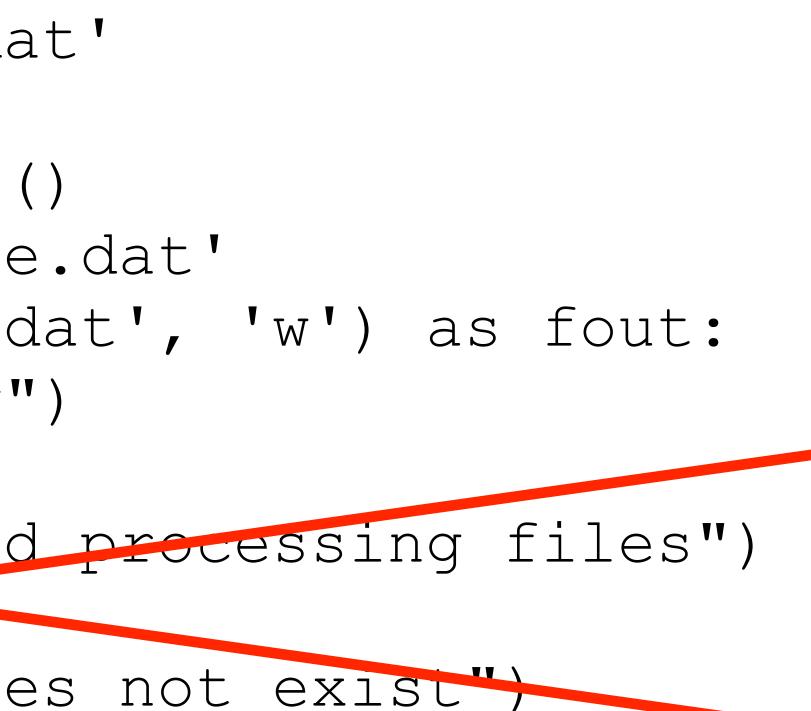
lines = f.readlines()

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)

• 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.
- 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
- try:

fname = 'missing-file.d with open(fname) as f: lines = f.readlines out fname = 'output-file with open ('output-file. fout.write("Testing except OSError as e: print(e.errno, e.filename, e)



