# Programming Principles in Python (CSCI 503/490)

### OOP & Exceptions

Dr. David Koop

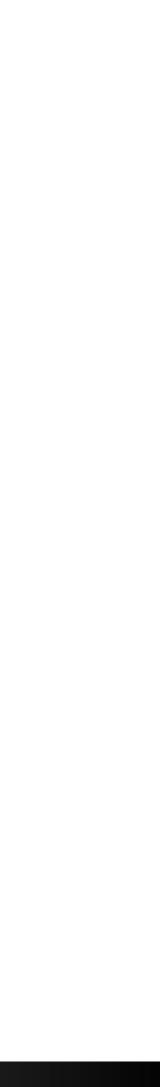




### D. Koop, CSCI 503/490, Fall 2024

## Quiz









- Which of the following instance variables is intended to be **private**?
  - (a) attr
  - (b) attr
  - (C) private:attr
  - (d) attr\_



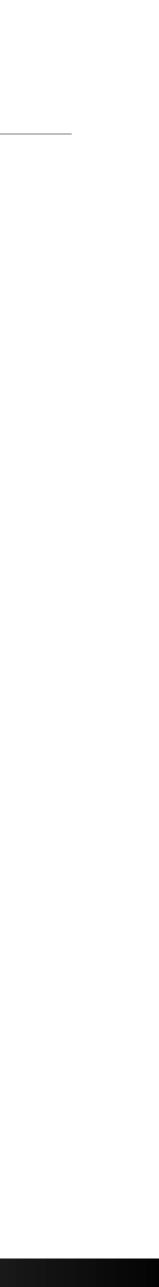






Which of the following is true?
(a) Python defines instance variables outside of methods
(b) Python uses the super method to access base class definitions
(c) Python does not allow multiple inheritance
(d) Python uses the extends keyword to declare a subclass





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









- Which decorator is used to define a **class method**? (a) \$class

  - (b) @class
  - (C) @classmethod
  - (d) @staticmethod









• Which method would be called to evaluate Square (4) + 8? (a) Square. radd (b) Square.\_\_add\_ (C) int. radd (d) int.\_\_add\_\_\_





7

## Inheritance

- Is-a relationship: Car is a Vehicle, Truck is a Vehicle Make sure it isn't composition (has-a) relationship: Vehicle has wheels,
- Vehicle has a steering wheel
- Subclass is specialization of base class (superclass) - Car is a subclass of Vehicle, Truck is a subclass of Vehicle
- Can have an entire hierarchy of classes (e.g. Chevy Bolt is subclass of Car which is a subclass of Vehicle)
- Single inheritance: only one base class
- Multiple inheritance: allows more than base class
  - Many languages don't support, Python does







# Instance Attribute Visibility Conventions in Python

- Remember, the naming is the convention (PEP8)
  - public: used anywhere
  - protected: used in class and subclasses
  - private: used only in the specific class
- You can still access private names if you want but generally **shouldn't**:
  - print(car1. color hex)
- Double underscores leads to **name mangling**:
  - self. internal vin is stored at self. Vehicle internal vin - This is why private makes sense (tied to defining class)







## Subclass

- Just put superclass(-es) in parentheses after the class declaration
- class Car(Vehicle):

. . .

- - self.num doors = num doors
- def open door(self):
- super() is a special method that locates the base class
  - Constructor should call superclass constructor
  - Extra arguments should be initialized and extra instance methods

### D. Koop, CSCI 503/490, Fall 2024

```
def init (self, make, model, year, color, num doors):
   super(). init (make, model, year, color)
```





10

## Overriding Methods

• class Rectangle: def init (self, height, width): self.h = heightself.w = weight def set height (self, height): self.h = height def area(self): return self.h \* self.w • class Square(Rectangle): def init (self, side): super(). init (side, side) def set height (self, height): self.h = heightself.w = height

- s = Square(4)
- s.set height(8)
  - Which method is called?
  - Polymorphism
  - Resolves according to inheritance hierarchy
- s.area() # 64
  - If no method defined, goes up the inheritance hierarchy until found









# Class and Static Methods

- Use @classmethod and @staticmethod decorators
- Difference: class methods receive class as argument, static methods do not
- class Square(Rectangle): DEFAULT SIDE = 10

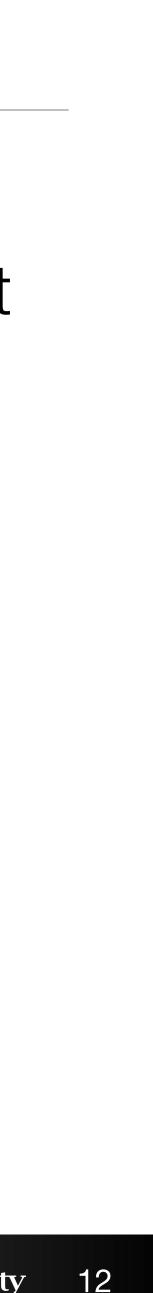
Qclassmethod def set default side(cls, s): cls.DEFAULT SIDE = s

@staticmethod def set default side static(s): Square.DEFAULT SIDE = s

### D. Koop, CSCI 503/490, Fall 2024

. . .





## Class and Static Methods

- class NewSquare(Square): DEFAULT SIDE = 100
- NewSquare.set default side(200) s5 = NewSquare()s5.side # 200
- NewSquare.set default side static (300) s6 = NewSquare()s6.side # !!! 200 !!!
- Why?
  - The static method sets Square.DEFAULT SIDE not the NewSquare.DEFAULT SIDE
  - self.DEFAULT SIDE resolves to NewSquare.DEFAULT SIDE

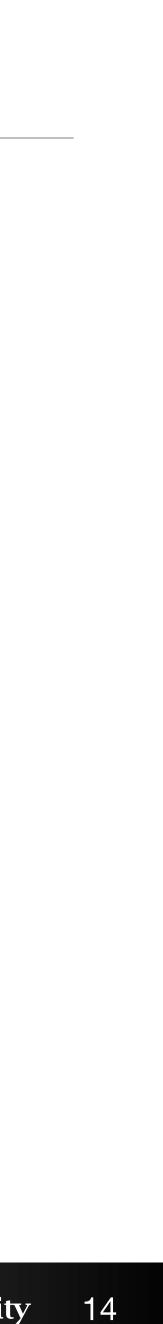




## <u>Assignment 5</u>

- Due next Monday
- Same Food data as A3
- Scripts, modules, packages
- Command-line program





### Interfaces

- In some languages, can define an abstract base class
  - The structure is defined but without implementation
  - Alternatively, some methods are defined abstract, others are implemented
- Interfaces are important for types
  - Method can specify a particular type that can be abstract
  - This doesn't matter as much in Python
- However, Python does have ABCs (Abstract Base Classes)
  - Solution to be able to check for mappings, sequences via isinstance, etc.
  - abc.Mapping, abc.Sequence, abc.MutableSequence







# 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







17

# 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]





## 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

### D. Koop, CSCI 503/490, Fall 2024

### • 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
- oriented programming (OOP)

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

### • With inheritance and overriding and polymorphism, we have true object-











### What if we just want to store data?







# 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







- 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 (nb\_mypy):

### • 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>



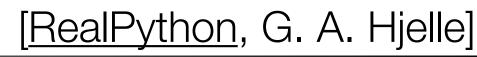






# Type Checking Pros & Cons

- 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 2024

### • 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 2024

## 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 2024

### • 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 2024

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







## Exception Granularity

- If you catch any exception using a k you may be masking code errors
- 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 2024

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





40