

Programming Principles in Python (CSCI 503/490)

Exceptions

Dr. David Koop

Quiz

Question 1

- Which of the following is **true**?
 - (a) Python does not support class methods
 - (b) Python defines instance variables outside of methods
 - (c) Python allows multiple inheritance
 - (d) Python uses the `extends` keyword to declare a subclass

Question 2

- Which decorator is used to define a **setter** for a property **year**?
 - (a) `@year(setter)`
 - (b) `@property(year)`
 - (c) `@year.setter`
 - (d) `@setter(year)`

Question 3

- Given a class `Car` that inherits from `Vehicle`, what should the first line of its constructor be?
 - (a) `super(make, model, color)`
 - (b) `super.__init__(make, model, color)`
 - (c) `Vehicle(make, model, color)`
 - (d) `super().__init__(make, model, color)`

Question 4

- Which of the following lines in a constructor defines an attribute intended to be **private**?
 - (a) `self._attr_ = 4`
 - (b) `self.__attr = 4`
 - (c) `self._attr = 4`
 - (d) `self.attr : private = 4`

Question 5

- Which method would be called to evaluate `4 + Square(8)` ?
 - (a) `Square.__add__`
 - (b) `Square.operator+`
 - (c) `Square.__radd__`
 - (d) `Square.operator__add`

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):
 def __init__(self, make, model, year, color, num_doors):
 super().__init__(make, model, year, color)
 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

# Overriding Methods

---

- ```
class Rectangle:  
    def __init__(self, height,  
                  width):  
        self.h = height  
        self.w = width  
  
    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 = height
 self.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
 ...

 @classmethod
 def set_default_side(cls, s):
 cls.DEFAULT_SIDE = s

 @staticmethod
 def set_default_side_static(s):
 Square.DEFAULT_SIDE = s
```

# 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`

# Assignment 5

---

- Due Monday
- Similar Pokémon entry data as A3, but **different** dataset
- Scripts, modules, packages
- Command-line program

# Types

# Checking type

---

- We can check the type of a Python object using the `type` method:
  - `type(6) # int`
  - `type("abc") # str`
  - `s = Square(4)`
  - `type(s) # Square`
- Allows comparisons:
  - `if type(s) == Square:`  
    `# ...`
- But this is **False**:
  - `if type(s) == Rectangle:`  
    `# ...`



# Checking InstanceOf/Inheritance

---

- How can we see if an object is an **instance** of a particular class or whether a particular class is a **subclass** of another?
- Both check is-a relationship (but differently)
- `issubclass(cls1, cls2)`: checks if `cls1` is-a (subclass of) `cls2`
- `isinstance(obj, cls)`: checks if `obj` is-a(n instance of) `cls`
- Note that `isinstance` is `True` if `obj` is an instance of a class that is a subclass of `cls`
  - `car = Car('Toyota', 'Camry', 2000, 'red', 4)`  
`isinstance(car, Vehicle) # True`

# 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):
 ...
```
- 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]`

# Mixins

---

- Sometimes, we just want to add a particular method to a bunch of different classes
- For example: `print_as_dict()`
- A mixin class allows us to specify one or more methods and add it as the second
- Caution: Python searches from left to right so a base class should be at the right with mixing



# 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)

[Deitel & Deitel]

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:
  - ```
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!

[[RealPython](#), G. A. Hjelle]

# 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 "abccabccabc"`
- 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 (nb\_mypy):
  -

# 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)

[[RealPython](#), G. A. Hjelle]



# 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

[[RealPython](#), G. A. Hjelle]



# 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

[[Java Tutorial](#), Oracle]

# 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

---

- 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!
- `try:`  
    `c = a / b`  
`except Exception:`  
    `c = 0`
- Can also have a **bare** except clause (matches any exception!)
- `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
- `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