Functional Programming and zip

Different Problem Decompositions

Programming languages support different problem decompositions.

- **Procedural:** programs are lists of instructions that tell a computer what to do
- **Declarative:** describe problem to be solved; language figures out how
- **Object-oriented:** program manipulates collections of objects; objects have internal state and methods that alter it

Functional Programming

- In functional programming style, the problem is decomposed into a set of functions.
  - Functions should take inputs, produce outputs, and have no internal state.
- Python supports functional programming (but also allows OO and procedural).
- The goal is to have *purely functional* functions that only use inputs to produce outputs and no other side effects.

Functional: Data Flows

- In functional style, the goal is to have data flowing through functions, not hidden away
- Benefits
  - ***Modularity:*** break apart problem into small pieces
  - ***Ease of testing:*** each function should have a clear relation between inputs and outputs
  - ***Composability:*** many general-purpose utility functions can be combined in new and interesting ways

Python Features for Functional

- Iterators and iterable: fundamental features of Python that make it easy to perform manipulations on elements of any sequence type
- Built-in data types that support iteration
  - Including file reading
- List comprehensions (create list object)
  - `l = [float(x) for x in range(100)]`
- Generator expressions (return iterator)
  - `l_iter = (float(x) for x in range(100))`

Python Features for Functional [2]

- Built-in functions
  - `map`: apply function to all elements of an iterable
  - `filter`: apply Boolean expression to all elements of an iterable
  - `enumerate`: count of elements in an iterable as a sequence of 2-tuples with the count from start and the underlying element
  - `sorted`: collects elements of the iterable in a list, sorts it, and returns it
  - `any(iter)`: returns `True` if any element is `True`
  - `all(iter)`: returns `True` if all elements are `True`

`zip` Function

```
l1 = list(range(5))
>>> l1
[0, 1, 2, 3, 4]
>>> itr = zip(l1, "abcdefghijklmnopqrstuvwxyz")
>>> list(itr)
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

- `zip` function merges multiple lists into a single list of tuples
- `zip` stops at the shortest list
- Not limited to two lists

`operator` Module

```
import operator

a = [ 1, 3, 5, 7 ]
b = [ 1, 2, 5, 9 ]

l = list(map(operator.eq, a, b))
print(l)
## [True, False, True, False]
```

- `operator` module defines all of Python's standard operators as functions
- Useful when programming in functional style as arguments to utilities

Itertools Module

`itertools` Module Highlights

The `itertools` module provides more functionality similar to `map`, `filter`, and `zip`.

- `filterfalse`: like `filter` function but keeps elements that evaluate to `False`
- `starmap`: iterable that computes the function using arguments from an iterable as tuples
- `zip_longest`: like `zip` but keeps going until all inputs are exhausted
- `islice`: approximates slicing for iterators

`filterfalse` Function

```
from itertools import filterfalse

L = [1, 2, 3, 4, 5, 6, 7]

## <-- gets evens
filter(lambda x: x%2==0, L)
## <-- also gets evens, since 0 evaluates to false
filterfalse(lambda x: x%2, L)
```

- `filterfalse` function returns an iterator containing all values from the input iterable where the function evaluates to `False`.
- This includes when values themselves evaluate to `False`.

`starmap` Function

```
from itertools import starmap
from collections import namedtuple

Point = namedtuple('Point', 'x y')
values = [ (100, 40), (200, 30), (400, 90) ]

## applies the function to the unpacking of each tuple
points = starmap(Point, values)
```

- `starmap` function returns an iterator containing all values from the function applied to input iterables of tuples
- Useful when values to apply are “pre-zipped”
    - Remember, `zip` can be used to join multiple iterables into a list of tuples

`zip_longest` Function

```
from itertools import zip_longest

s1 = 'abcdef'
s2 = 'ghi'

for a, b in zip_longest(s1, s2):
    print(a, b)

for a, b in zip_longest(s1, s2, fillvalue='-'):
    print(a, b)
```

- `zip_longest` function returns an iterator over a sequence of tuples formed by combining elements from each input iterable
- Keeps going until all inputs are exhausted

`islice` Function

```python
from itertools import islice

## only prints the first 5 values of the range
for i in islice(range(0, 10), 5):
    print(i)

## skips the first 5 values of the range
for i in islice(range(0, 10), 5, None):
    print(i)

## gets lines 3-10 of a file with their line number
with open(filename) as f:
    for lineno, line in enumerate(islice(f, 2, 10), start=3):
        print(f'line {lineno} :', line[:-1])
```

- Iterators cannot be sliced as actual sequences.
- `islice` function allows start, stop, and step to be specified.

functools Module

`functools` Module

Provides higher-order function capabilities

- Partial function application
  - Creating a new function out of an existing function by filling in a parameter
- Reduction
  - Repeated function application over the entire iterable

`partial` Function

```
from functools import partial
import operator

a = [ 1, 3, 5, 7, 9]

## greater than 5 function
## creating a function that performs (5 < x)
## returns true when the argument is greater than 5
gt_5 = partial(operator.lt, 5)

list(map(gt_5, a))
## [False, False, False, True, True]
```

- `partial` creates a new function out of an existing function by filling in parameter arguments
- Can fill in both positional parameters (in order) and keyword parameters

`reduce` Function

```
from functools import reduce

b = [ 'abc', 'a', 'abcd' ]

def is_longer(curr_max, value):
    if len(value) > curr_max:
        return len(value)
    else:
        return curr_max

longest_str_len = reduce(is_longer, b, 0)
```

- `reduce` takes a function with two parameters and returns a single value
- Initially passes the first two elements, then the result of that call and the third element, and so on
- For four elements: `f(f(f(A,B), C), D)`
- Initial argument can replace ‘A’ in the call above
- Simple loop is often probably faster

Object-Oriented Programming

Object-Oriented (OO) Programming

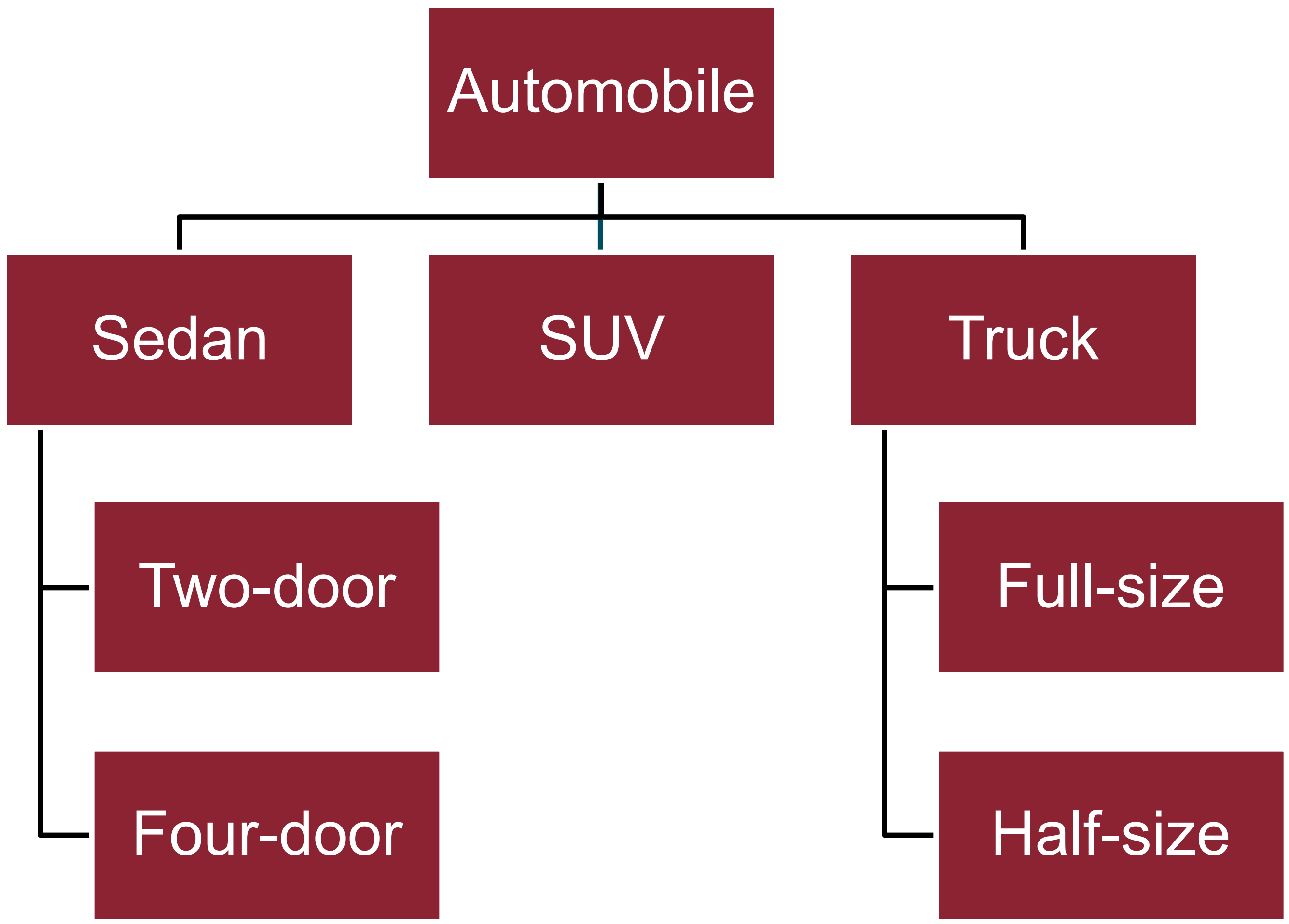
In some sense, the “opposite” of functional programming

- Classes encapsulate both state and methods that change that state
- System functionality comes about due to method calls between objects; state is hidden and protected
- Very popular; allows developers to model the functionality in a natural way

O-O Benefits

- Classes can be developed and tested independently, then composed.
- General-purpose classes can be reused.
- Class hierarchy matches well with many real-world systems and allows code to be shared/overridden in a natural manner.
- Python supports O-O very well (but does not force it).

Class Hierarchy



Class Hierarchy [2]

- Domain of interest is decomposed into "classes," which are categories of interest.
- Moving from root to leaf, classes become more specialized.
- Class design involves trade-offs (could have decomposed based on make).
- Functionality (attributes/methods) implemented higher in the hierarchy can be reused by more classes.

OOP

Defining Classes

Basic Class Definition

```
class Point3D:
    """Represents a point with 3 coordinates (x, y, z)."""

    def __init__(self, x, y, z):
        """This is the class constructor."""
        self.x = x
        self.y = y
        self.z = z

## >>> p1 = Point3D(10, 11, 4)
## >>> print(p1.x, p1.y, p1.z)
## 10 11 4
```

- The code above defines a new type called `Point3D`.
- The constructor (called `__init__` in Python) takes the `self` reference and three parameters.
- The class name works like a factory function that returns a new instance of the class, having called the constructor.

Basic Class Definition [2]

- Each call to the class object (defined with the class keyword) creates a new object of that type.
- These are called *instances.*
- Attributes (data attributes and methods) defined on instances are referenced from within the object by `self`.
  - They are referenced from outside the object by the variable referring to the object.
- Method objects (member functions) are called with the same syntax and always take a reference to `self` as the first parameter.

Class With a Method

```
class Point3D:
    EPSILON = 1e-10
    def __init__(self, x, y, z):
        """This is the class constructor."""
        self.x = x
        self.y = y
        self.z = z

    def is_on_x(self):
        return abs(self.x) < Point3D.EPSILON

## >>> p1 = Point3D(10, 11, 4)
## >>> p1.is_on_x()
## False
```

- `EPSILON` is a variable defined on the class and accessed via class name.
- `is_on_x` is a method. When called on the object, `self` is implicitly passed in as the first parameter.
- Methods use `self` to access instance data.

Magic Methods

- “Magic methods” are simply methods that may be defined by a class and are automatically called in certain common situations.
- `__init__` on the previous slides is a good example—it is called after an object is created to initialize the attributes.
- Magic methods are used to make our classes interact with the Python system in natural ways.
- They are also known as “dunder” methods (double-underscore).

__repr__ and __str__

```
class Point3D:
    def __init__(self, x, y, z):
        """This is the class constructor."""
        self.x = x
        self.y = y
        self.z = z

    ## returns the 'official' string representation
    def __repr__(self):
        return "Point3D(%d,%d,%d)" % (self.x, self.y, self.z)

    ## returns a 'convenient' string representation
    ## if not defined, __repr__ will be used
    def __str__(self):
        return "(%d,%d,%d)" % (self.x, self.y, self.z)


## >>> p1 = Point3D(10, 11, 4)
## >>> repr(p1)
## 'Point3D(10,11,4)'
## >> print(p1)
## (10,11,4)
```

Inheritance

Inheritance

Classes can “extend” base classes.

- Defining a more specialized version of a base class
- Adding attributes and methods
- Overriding methods on the base class with more specialized versions

Inheritance Example

```
class Animal:
    def __init__(self, age, weight):
        self.age = age
        self.weight = weight

    def __str__(self):
        return f'an animal aged {self.age} and weighing {self.weight} pounds'

## this creates a Cat class, which is a subclass of Animal
class Cat(Animal):
    pass

## this creates a Dog class, which is also a subclass of Animal
class Dog(Animal):
    pass

## these call Animal.__init__
dog = Dog(10, 30)
cat = Cat(10, 11)

## these call Animal.__str__
print(dog)
print(cat)
```

Checking Types

```
## checking types
assert type(cat) == Cat
assert type(dog) == Dog

## using isinstance to check for subclasses
assert isinstance(cat, Cat)
assert isinstance(cat, Animal)
assert isinstance(cat, object)
assert isinstance(dog, Animal)
assert not isinstance(dog, Cat)
```

- `type` returns the actual (instantiated) type of an object
- `isinstance` checks if an object is of the named type or derived from it (recursively)

Adding Methods

```
class Cat2(Animal):
    def meow(self):
        print('Meow!')

class Dog2(Animal):
    def bark(self):
        print('Woof!')

## each still inherits from the Animal class...
cat = Cat2(10, 15)
dog = Dog2(6, 45)

## ...but has additional methods available
cat.meow()
dog.bark()
```

Methods on base class are still available.

Adding Attributes

```
## simple, but repetitive
class Pet(Animal):
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def __str__(self):
        return f'{self.name}, an animal aged {self.age} and
        weighing {self.weight} pounds'

## better! using functionality of base class via super()
class Pet2(Animal):
    def __init__(self, name, age, weight):
        super().__init__(age, weight)
        self.name = name

    def __str__(self):
        return f'{self.name}, {super().__str__()}'
```

Defining and Handling Exceptions

Exception Hierarchies

Exceptions are defined using inheritance.

- Exception is the base class of all exceptions.
- Specific exceptions are derived from this.
- Programmers can choose where in the hierarchy they handle an exception.

Handling Exceptions

```
try:
    i = int(input('Enter a number: '))
    print('Your number plus one is:', i + 1)
except ValueError:
    print('That was not a number...')
except Exception:
    print('Something else went wrong...')
```

- `ValueError` is derived from `Exception`
- First `except` clause handles `ValueError` (and any exceptions derived from it)
- Second except clause catches any exceptions that are not derived from `ValueError`

Exception Handling Order

```
try:
    i = int(input('Enter a number: '))
    print('Your number plus one is:', i + 1)
except Exception:
    print('Something else went wrong...')
except ValueError:
    print('That was not a number...')
```

- Will not work as expected
- First `except` clause will handle all exceptions
  - Exception base class will match
- Second `except` clause will never execute

User-Defined Exceptions

```python
## Define a custom exception
class UserDefinedException(Exception):
    """User-defined Exception."""
    pass

## Define something even more specific
class MoreSpecificException(UserDefinedException):
    """Exception used in a more specific situation."""
    pass
```

- Defining program-specific exceptions is simply a matter of extending the Exception class
- `pass` syntax allows for an empty class definition
    - Statement that does nothing
- These inherit all attributes and methods from base class