Coding Bootcamp Code in Python

OBJECT-ORIENTED PYTHON

Object-orientation

- Python types are classes
  - e.g., `(14).bit_length()` == 4
    - `14` is an object of class `int`
    - `bit_length` is object method defined in class `int`

You are using objects all the time!

- Objects of simple Python types are immutable
  - Operations/methods instantiate new objects

Value versus object identity

- Simple Python types
  - Value identity: `(14 == 14) == True`
  - Object identity: `(14 is 14) == True`
    - However, Python version dependent!
- Other Python types, general classes
  - e.g., two `set` objects:

    `a = {'alpha'},b = {'alpha'}`
    - Value identity: `(a == b) == True`
    - Object identity: `(a is b) == False`

Defining your own classes

- Class definition:

```
class Point:
    …
```

- Objects are instances of classes
  - instantiated by calling constructor
  - have
    - attributes
    - methods
- Classes have
  - attributes
  - methods

A simple point...

```
from math import sqrt

class Point:

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def distance(self, other):
        return sqrt((self.x - other.x)**2 +
                    (self.y - other.y)**2)

    def __str__(self):
        return f'({self.x}, {self.y})'
```

constructor for `Point` objects

method to compute distance

creates string representation for `Point` object

Making a point… or two

```
…
def main():
    p = Point(3, 4)
    q = Point(-2, 5)
    print(p.x, p.y)
    print(p, q)
    print(p.distance(q))
    p.x = 12.3
    print(p)…
```

create `Point p` at 3, 4

create `Point q` at -2, 5

access `p`'s x- and y-coordinates

calls `__str__` method indirectly on `p` and `q`

compute distance from `p` to `q`

modifying `p`

```
$ python point_driver.py
3.0 4.0
(3.0, 4.0) (-2.0, 5.0)
5.0990195136
(12.3, 4.0)
```

`distance` method invoked on `Point p`, with `Point q` as argument

More to the point…

- What if points should not be moved?

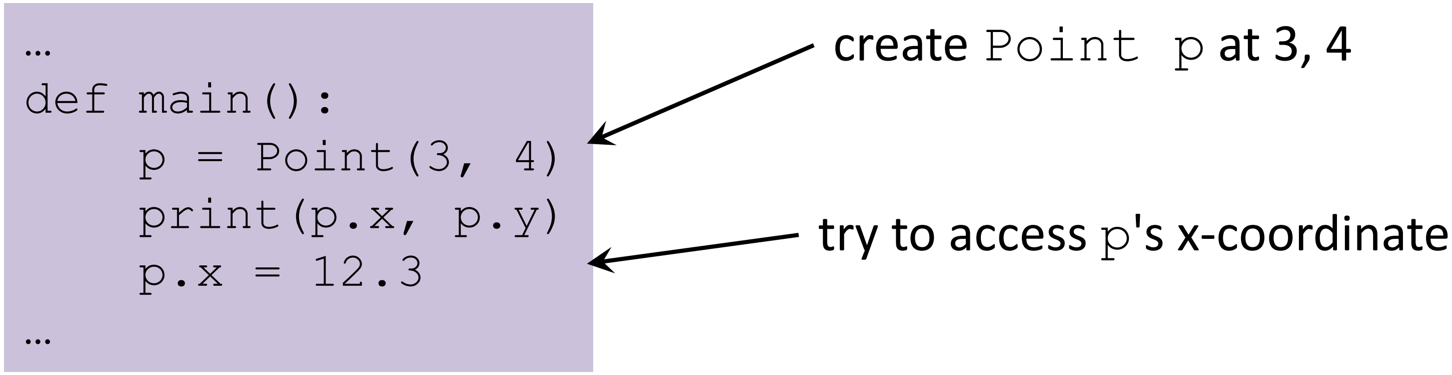
```
class Point:

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y
    …
```

constructor for `Point` objects

getter for object's `__x` attribute

getter for object's `__y` attribute

Making a definite point



```
…
def main():
    p = Point(3, 4)
    print(p.x, p.y)
    p.x = 12.3
…
```

create `Point p` at 3, 4

try to access `p`'s x-coordinate

```
$ python point_driver.py
3.0 4.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Object attributes

- Make object attributes "private" by hiding them, by convention, use __ prefix

```
self.__x = x
```

- Create getter/setter method to control access to object attributes

```
@property
def x(self):
    return self.__x
```

Determine object's state

Object attribute can not accidently be modified, i.e., read-only

Object attributes: control

- Getter, but no setter

```
…
def main():
    p = Point(3, 4)
    print(p.x)
    p.x = 4.4
    print(p.x)
…
```

Protects against modification of read-only attributes

```
$ python point_driver.py
3.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Object attribute: setter

- Implementing setters improves control, assignment to attribute is "intercepted" by setter method

```
class Point:
…
    @x.setter
    def x(self, value):
        self.__x = float(value)
…
```

E.g., ensures proper type conversion:
`p.x = 3` results in `float`, not `int` for `__x` attribute

Non-trivial getter/setter

- Derived attribute: coordinates as 2-tuple

```
class Point:
…
    @property
    def coords(self):
        return (self.x, self.y)

    @coords.setter
    def coords(self, value):
        self.x = value[0]
        self.y = value[1]
…
# Use coords getter/setter
print(p.coords)
p.coords = (3.5, 7.1)
```

returns a 2-tuple

2-tuple as argument

More object methods

```
from math import sqrt, isclose

class Point:
    …
    def on_line(self, p, q, tol=1.0e-6):
        if not isclose(p.x, q.x, abs_tol=tol):
            a = (q.y - p.y)/(q.x - p.x)
            b = p.y - a*p.x
            return isclose(self.y, a*self.x + b, abs_tol=tol)
        else:
            return isclose(self.x, p.x, abs_tol=tol)
…
# check whether r is on line defined by p and q
if r.on_line(p, q):
    …
```

Python 3.5+

`on_line` **method invoked on** `Point r`**, with** `Point p` **and** `q` **as argument**

Object methods

- Used to
  - retrieve information on object
  - modify or manipulate object
  - derive information from object with respect to other objects
  - …

Determine what objects can do, or can be done with

Static methods

```
…
class Point:
    …
    @staticmethod
    def all_on_line(p, q, *points):
        for r in points:
            if not r.on_line(p, q):
                return False
        return True
…
# check whether p, q, r, v and w are on a line
if Point.all_on_line(p, q, r, v, w):
    …
```

`all_on_line` method invoked on `Point` class with `Point` `p`, `q`, `r`, `v`, `w` as arguments, class ignored

Variable length argument lists

- Arbitrary positional arguments: `*argv`

```
@staticmethod
def all_on_line(p, q, *points):
    for r in points:
        if not r.on_line(p, q):
            return False
    return True
```

arguments

available as tuple

- Arbitrary keyword arguments: `**argv`
  - Available as dictionary

Note: not specific to object oriented programming

More elegant solution

- Semantics: `True` if `True` for all elements in points

```
@staticmethod
def all_on_line(p, q, *points):
    for r in points:
        if not r.on_line(p, q):
            return False
    return True
```

- More elegant: `all`(…)

```
@staticmethod
def all_on_line(p, q, *points):
    return all(r.on_line(p, q) for r in points)
```

- Similar: `any`(…)

Quick interlude

- What attributes/methods does a class have?

```
>>> from point import Point
>>> p = Point(3.7, 5.1)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_Point__x', '_Point__y',
 'all_on_line', 'coords',
 'distance', 'on_line', 'x', 'y']
```

Inheritance

- Class can extend other class
- For Python 2: make classes inherit from `object`, ensure they can be extended later:

  ```
  class Point(object):
  ```
- New class inherits attributes & methods from parent class
- New class can implement new methods, define new attributes
- New method can override methods of parent class
- New class can inherit from multiple parent classes

Points with mass

```
class PointMass(Point):

  def __init__(self, x, y, mass):
    super().__init__(x, y)
    self.__mass = float(mass)

  @property
  def mass(self):
    return self.__mass

  def __str__(self):
    return '{0}: {1}'.format(
      super().__str__(),
      self.mass)
```

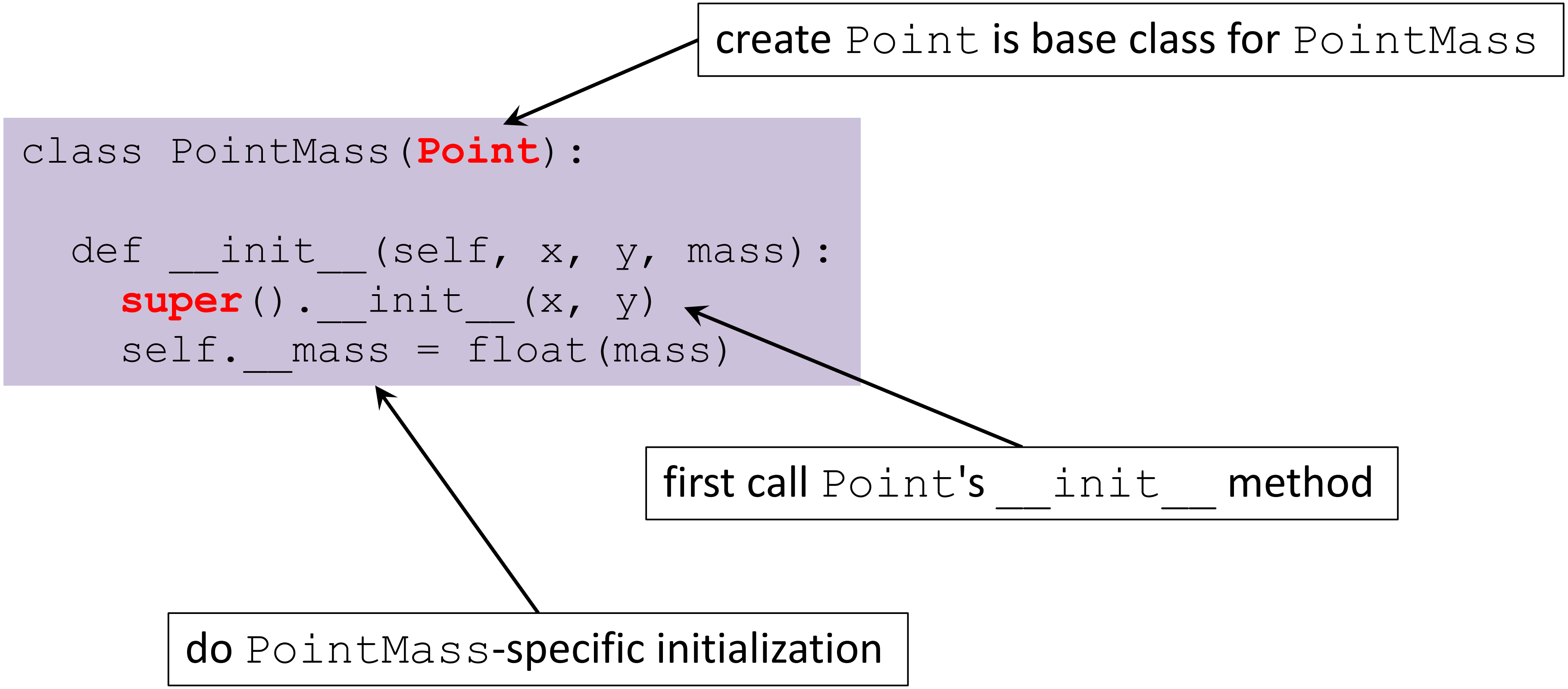
constructor of `Point` overridden

new object method

`str` method of `Point` overridden

`PointMass` objects have `x`, `y`, `distance`, `on_line` methods as well
`PointMass` class has `all_on_line` methods

Base classes & derivation



```
class PointMass(Point):

  def __init__(self, x, y, mass):
    super().__init__(x, y)
    self.__mass = float(mass)
```

create `Point` is base class for `PointMass`

first call `Point`'s `__init__` method

do `PointMass`-specific initialization

Point with mass is still Point

```
def main():
    p = PointMass(3, 4, 1)
    q = Point(-2, 5)
    print(p.x, p.y, p.mass)
    print(p.distance(q))
```

create `PointMass p` at 3, 4 and mass 1

create `Point q` at -2, 5

`p` is a Point, so has `distance` method

```
$ python point_driver.py
3.0 4.0 1.0
5.09902
```

Class attributes

```
class PointMass(Point):

    __default_mass = 1.0

    def __init__(self, x, y, mass=None):
        super().__init__(x, y)
        if mass is not None:
            self.__mass = float(mass)
        else:
            self.__mass = PointMass.__default_mass
    …
    @classmethod
    def set_default_mass(cls, mass):
        cls.__default_mass = float(mass)
```

class variable `__default_mass`

setter for class' `__default_mass` attribute

Determine state of class

All those methods

- Object methods
  - work on individual objects
  - take object as first argument (`self`)
- Class methods
  - `@classmethod`
    - work at class level
    - take class as first argument (`cls`)
  - `@staticmethod`
    - work at class level
    - ignores object or class it is called on

Code Pack 06

- A. Python fundamentals:
- ~~1. Primitive Datatypes and Operators~~
- ~~2. Variables and Collections~~
- ~~3. Control Flow and Iterables~~
- ~~4. Functions~~
- ~~5. Modules~~
- 6. Classes

Coding Bootcamp Code in Python

GETTING THINGS IN AND OUT: I/O & COMMAND LINE ARGUMENTS

Reading lines from file handles

- Standard file handles:
  - `sys.stdin`: standard input (keyboard, pipe in)
  - `sys.stdout`: standard output (screen, pipe out)
  - `sys.stderr`: standard error (screen, pipe out)
- Reading a single line:
  - `sys.stdin.readline()`: returns `str`
- Reading all lines at once:
  - `sys.stdin.readlines()`: returns `list` of `str`

Note: line endings, e.g., `\n` or `\r\n` are included

Note: `readline()`, `readlines()` are methods on file handles

Reading & memory consumption

- Remember, `readlines()` method reads whole file at once
  - For large files, creates long list = lots of memory
- Avoid:

```
…
for line in sys.stdin.readlines():
    …
```

- Use:

```
…
for line in sys.stdin:
    …
```

Returns iterator, not `list`
Memory friendly!

Writing to file handles

- `print` function writes objects to `sys.stdout`, adds `'\n'` (or `'\r\n'`) and applies `str()` conversion function by default
- `write(…)` method writes `str` to file handle, e.g.,
  - `sys.stderr.write('### error: number is negative\n')`
  - `sys.stdout.write(output_str)`
- `flush()` method flushes output to disk
  - At least, tells OS to do so

More on print

- `print` has some useful optional arguments
  - `file`: allows to print to any open file handle, e.g., `sys.stderr` (default: `sys.stdout`)

    ```
    print("# error: number should be positive",
          file=sys.stderr)
    ```

  - `sep`: character to separate multiple objects to print (default: `' '`), e.g.,

    ```
    print('alpha', 3, 5.7, sep='\t')
    ```

  - `end`: character to add when all arguments are printed (default: `'\n'`), e.g.,

    ```
    print('next print will be on same line',
          end='')
    ```

  - `flush`: whether to combine print with a flush on the file handle (default: `False`),

    ```
    print('read done', file=sys.stderr,
          flush=True)
    ```

Simple command line arguments

- Script name & command line arguments in `sys.argv`

```
import sys

if __name__ == '__main__':
    print(sys.argv)
```

```
$  python  cla_printer.py
['cla_printer.py']
$  python  cla_printer.py  alpha  beta  3.5
['cla_printer.py', 'alpha', 'beta', '3.5']
$ python  cla_printer.py  'alpha beta'  3.5
['cla_printer.py', 'alpha beta', '3.5']
```

Note:
all values
are `str`

Okay for very simple cases, better: use argparse

Code Pack 07

A. my_repl.py
B. bot_create_a_story.py
C. distance.py

Coding Bootcamp Code in Python

WRITING DOCUMENTATION & SIMPLE TESTING

Writing documentation

- Documentation is very important!
  - use DocString

```
def parse_line(line, sep=None):
    '''Split a line into its fields, convert to the
       appropriate types, and return as a tuple.'''
# using \r, \n should work for Windows & *nix
    data = line.rstrip('\r\n').split(sep)
    return (int(data[0]), int(data[1]), float(data[2]))
```

```
>>> import data_parsing
>>> help(data_parsing.parse_line)
Help on function parse_line in module validator:

parse_line(line)
    Split a line into its fields, convert to the
    appropriate types, and return as a tuple
```

Formatting docstrings

```
def parse_line(line, sep=None):
    '''Split line into fields,
       converted to appropriate types

    Parameters
    ----------
    line: str
        line of input to parse
    sep: str
        field separator, default
        whitespace

    Returns
    -------
    tuple (int, int, float)
        data fields: case number,
        dimension number, temperature
    '''
    …
```

numpy/scipy

Many options

- Google
- reStructured Text
- numpy/scipy

What to document and how?

- DocString for
  - functions
  - classes
  - methods

  see later

  - modules
  - packages
- Comments
  - particular code fragments you had to think about

Assertions

- Testing pre and post conditions
  - Programming by contract

```
def fac(n):
    assert type(n) == int, 'argument must be integer'
    assert n >= 0, 'argument must be positive'
    if n < 2:
         return 1
    else:
        return n*fac(n - 1)
```

Optional

```
$ python  -c 'from fac import fac; print(fac(-1))'
…
assert n >= 0, 'argument must be positive'
AssertionError: argument must be positive
```

Assert use cases

- For development only, *not* production!
- *Not* a substitute for error handling, i.e., exception handling
- Run without assertions, run optimized: –O

```
$ python  -O  -c 'from fac import fac; print(fac(-1))'
1
```

Useful feature, but don't abuse!

Testing: meeting expectations

- Tests are important!
  - unittest: more features, but harder
  - doctest: simple

> *A program that has not been tested does not work.*
>
> — Bjarne Stroustrup

Statement to execute

```
def parse_line(line):
    '''Split a line into its fields, convert to the
       appropriate types, and return as a tuple.
       >>> parse_line('5  3  3.7')
       (5, 3, 3.7)
      '''
    data = line.rstrip('\r\n').split()
    return (int(data[0]), int(data[1]), float(data[2]))
```

Expected result

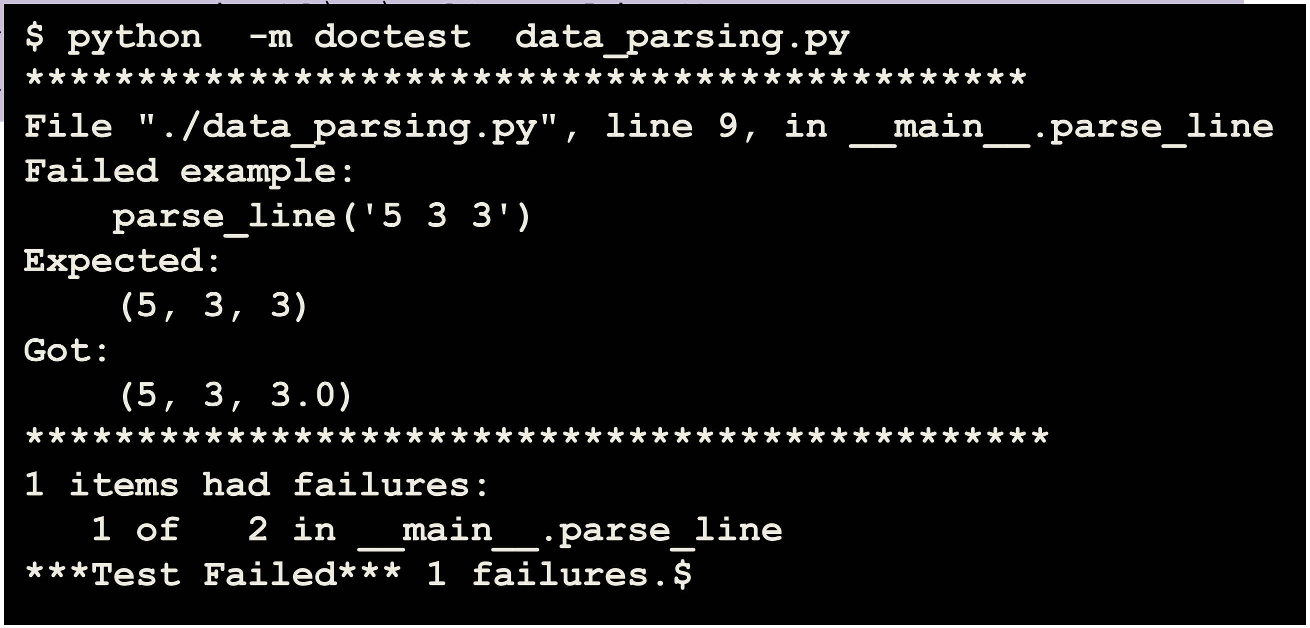
- Run tests

No output: hooray, all tests passed!

```
$ python  -m doctest data_parsing.py
$
```

Failing tests

```
def parse_line(line):
    '''Split a line into its fields, convert to the
       appropriate types, and return as a tuple.
       >>> parse_line('5  3  3.7')
       (5, 3, 3.7)
       >>> parse_line('5 3 3')
       (5, 3, 3)
     '''
    data = li
    return (i
```



```
$ python  -m doctest  data_parsing.py
*********************************************
File "./data_parsing.py", line 9, in __main__.parse_line
Failed example:
    parse_line('5 3 3')
Expected:
    (5, 3, 3)
Got:
    (5, 3, 3.0)
**********************************************
1 items had failures:
   1 of   2 in __main__.parse_line
***Test Failed*** 1 failures.$
```

Code Pack 08

A. Create and document mymath.py