CSC148
Introduction to Computer Science

Danny Heap

Contents

introduction

Computers are to human beings as sand is to oysters: they irritate us into producing beautiful pearls. Boundless promise of machines automatically carrying out our instructions for solving problems is reined in by the constraints of computing devices and human organization. Tension between promise and constraint leads us to form careful observations and rules about how we work with these wonderful automatic problem-solving devices. We evolve from hackers into computer scientists.

We remain true to our hacker intuition: when we see code examples, we try them out, modify them, and try to improve them. Remember that when you see code fragments in these notes. We add our scientific rigour: we try to understand causes and effects, we work out rules to make our code development faster and less error-prone, then we test and reconsider those rules.

We work in a community of computer scientists, so we test our own understanding against other critical minds, and we benefit from discoveries in the past.

Welcome to computer science.

Chapter 1

abstraction

Computer scientists need to emphasize some details, and suppress others, in order to understand computations. You've already seen this as a programmer — you give meaningful names to functions and data, rather than repeating the gory details of how you compute them. The user of your code deals with the high-level — function name, and the value of its arguments — rather than low-level details of how it works.

1.1 objects

Most modern computers store and manipulate data as **bi**nary digi**ts**, represented by bits — strings of 0s and 1s, high and low voltage, or magnetic north and south dipoles. Luckly, we don't often deal with raw bits when working with computer programs: the bits are grouped into meaningful chunks that represent number, strings, images, toasters, and other program components. We'll call these chunks **objects**.

Every value in a Python program is an object and has a (virtual) memory address. You can check this using **id**, or **is** (try typing these in a Python interpreter):

```
id(5)
id("five")
5 is "five"
```

Two **separate** objects may represent the **same** value. Since the values may be computed in independent ways, Python may not realize that it could use just one object to represent both:

```
>>> w1 = "words"
>>> w2 = "swords"[1:]
>>> w1 == w2
>>> w1 is w2
```

Objects can contain both information and behaviour. We get at these using ".", the dot operator. Turtle objects, used for drawing, provide an example (type these in a Python interpreter):

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.DEFAULT_ANGLEOFFSET
>>> t.forward(100)
```

turtle.Turtle is a class, or blueprint, for creating turtles. Python allows you to add new parts to an existing class, simply by assigning to a name. If you try typing the example below, you can add wings to turtles, on the fly:

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.wings
>>> turtle.Turtle.wings = "waterproof feathers!"
>>> t.wings
```

1.2 abstract data types (ADTs)

Objects, with their information and behaviour, can represent entities that we wish to store and manipulate to solve problems, usually on a computer. It is often useful to focus on the **semantics** (the meaning of the real-world entity being represented) rather than the details of how this is implemented. There are at least two reasons this is an advantage:

1. We can think about algorithms, or recipes, for solving problems more freely if we don't have to include all the details of how our objects are implemented.

2. Details of how objects, and their components, are stored and accessed vary between programming languages, whereas a really good algorithm can be translated into any programming language.

These are the motivations for **Abstract Data Types (ADTs).** An ADT specifies the intended meaning of the data it stores, and the operations it provides on that data. It does not talk about the how to store and manipulate the data in a particular programming language.

Integers are an example of an abstract data type. We don't fuss about how our computer represents integers 257 or 133 when we add them, and we expect a programming language to provide some way to evaluate

```
257 + 133.
```

Integer ADT represents positive and negative whole numbers, as well as 0. Integers also include behaviour: addition, multiplication, division, subtraction, and comparisons such as $<, >, ==$. By hiding the details of how integers are represented and support these operations, we can focus on interesting algorithms that use integers.

Stacks provide another example of an ADT. Stacks can be implemented in all computer languages, and in the real world — you may have several on your shelves or desk. A **stack**

- stores items
- provides an operation to remove (**pop**) the top item
- provides an operation to add (**push**) a new item onto the top of the stack
- provides a way to tell whether the stack is empty (**is_empty**)

1.2.1 hide details

Notice that the bullet list in the previous section said nothing about the concrete details of how a **stack** is implemented. The items could be dishes on your shelf, held together by gravity and friction, the **push** and **pop** methods being implemented by your body, and the **is_empty** method being implemented by your eyes and visual cortex.

Or a **stack** could be implemented in Python as a **list**, where **push** and **pop** add items to the end of the list, and **is_empty** uses some Python code to check the length of the list and see whether it is 0.

In a **stack** ADT, we don't consider these grubby details, because we're more interested in what sort of things we can do with a stack. There are many algorithms that use **stacks**, and we don't want to obscure the deep thinking about algorithms by worrying about the low-level implementation.

Here's a simple problem that you can probably solve with a **stack**. Given a sequence of items, produce the reverse of that sequence, using a **stack** and its operations **push**, **pop**, and **is_empty** (probably more than once).

1.2.2 expose interfaces

When you implement an ADT in your favourite programming language, you will want it to be available for others (and yourself!) to use. You'll need to tell the users (clients) of your code what it does. This means that your documentation will specify an **interface** that tells clients what operations your ADT is capable of. Here's an example of a python **docstring** for a **stack**:

```
"""
Stack ADT.

Operations:
    pop(): remove and return top item, provided
         stack is non-empty
    push(item): store item on top of stack
    is_empty(): return whether stack is empty.
"""
```

In addition to this class docstring, you would have a docstring for each of the methods in your class. Taken together, these provide a public interface for the stack ADT.

Once you have published an interface, it becomes incredibly awkward to modify it. Imagine billions of enthusiastic clients are already using your ADT. That means they like it, and have written their own code that depends on it. If sometime next week you change the behaviour of, say, **push**, you create two issues. First, you have to inform your clients of the change — they may not notice that your **docstring** has changed with your code update. Second (and most serious) your clients may need to re-write all their code that used your ADT. They won't be happy.

This means you should carefully design and test your ADT before you publish an interface. Then you should stick to that interface! There are techniques for **extending** an interface (see **inheritance** below, as well as composition).

1.2.3 bundle information and behavior

Notice that the **stack** ADTs combines information (the stored data) with operations (**push**, **pop**, and **is_empty**). You could certainly get the same results by having a separate structure (e.g. a list) and some functions that use it. However, this burdens your clients with the task of keeping track of the separate pieces — information and behaviour — in their own code.

An important feature of ADTs is that they combine information (implemented as attributes in Python, see below) and behaviour (implemented as methods in Python, also see below).

1.3 concretize ADTs with classes

An ADT doesn't specify how you implement it in a particular programming language. In Python, types of similar data are specified by **classes** — you've used built-in classes such as **str** or **list**.

If you invoke

```
>>> help(str)
```

. . . at the Python command-line, you'll be able to read documentation on what the ADT **str** represents, and what operations are available for **str**. All of this is part of the code, written by some intrepid Pythonista, deep in the bowels of your Python distribution. When you use Python, you take advantage of the work that went into creating this built-in class representing sequences of characters, and the operations that class **str** supports.

Eventually you'll want a class that isn't already implemented. A class declaration creates a new class for creating objects of that type.

1.3.1 user-defined classes

You'll first need to identify the parts, the **attributes** and **behaviour** of your new class. Below is an example from How to think like a computer scientist. Identify the most important **noun** — a good candidate for your class **name**. Then identify **verbs** the noun tends to use — good candidates for behaviour, or **operations**, of your class. There may be some less-important nouns — good candidates for **attributes** of your class.

> In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

deeply wrong class implementation

We can build the **Point** class in a rough-and-ready way, to get an idea of how the class mechanism works. However **don't try this at home**, or at least when you continue to declare classes in your subsequent work.

The minimum you need is to tell Python your new class name. We start out with a class that has no methods or attributes:

```
>>> class Point:
...     pass
...
```

Of course, we could add **x** and **y** coordinates directly to a point **p**, using something like **p.x = 5**. But we're programmers, so we can write a function (not part of our class... yet):

```
>>> def __init__(point, x, y):
...     point.x, point.y = float(x), float(y)
...
```

We'll explain the strange naming that uses double underscores later. Now, if **p** is a point, then **__init__(p, 3, 5)** will give it x-coordinate 3 and y-coordinate 5.

The distance from the origin of point (x, y) is given by the formula $\sqrt{x^2 + y^2}$. Of course, there is a **sqrt** function in the module **math**, but you can do the same thing by raising to the power 1/2. Here's an implementation of a function that returns the distance of a point from the origin (assuming the point has attributes **x** and **y**):

```
>>> def distance_from_origin(point):
...     return (point.x**2 + point.y**2)**(1/2)
...
```

You now have enough machinery in place to create a point, give it coordinates, and find its distance from the origin:

```
>>> p = Point()
>>> __init__(p, 3, 4)
>>> distance_from_origin(p)
5.0
```

It's inconvenient to have the functions **_init_** and **distance_from_origin** separate from the class declaration for **Point**. We can add them to class **Point** as methods — basically functions that are part of each instance of **Point**. Every class expects to have a standard function called **_init_**, so we can use our **_init_** function for that:

```
>>> Point.__init__ = __init__
>>> p2 = Point(12, 5)
>>> p2.x
12
>>> p2.y
5
>>> Point.__init__(p2, 3, 4)
>>> p2.x
3
```

Notice there are **two** ways of using method **_init_**. You can access it using the name of the class, **Point**, and provide it with values for a **Point**, and the desired **x** and **y** coordinates — exactly the way **_init_** works, just using a different name for the function.

The other way to access **_init_** happens automatically whenever you create a new point, since **_init_** is a standard special method. There is no need to provide a particular **Point** value, since it's obvious that we're referring to the one being created.

Similarly, we can add **distance_to_origin** to class **Point**:

```
>>> Point.distance_from_origin = distance_from_origin
>>> p3 = Point(12, 5)
>>> p3.distance_from_origin()
13.0
>>> Point.distance_from_origin(p3)
13.0
```

Notice that when we access **distance_from_origin** using the name of **Point p3**, there's no need to provide it with the first argument specifying a **Point**. However, if we access it using the name of class **Point**, we need to tell it a name for a particular point.

Notice the different number of parameters: **Point.distance_from_origin(...)** takes one parameter, a reference to a **Point**. On the other hand **p3.distance_from_origin()** takes zero parameters, since it already has the name of **Point** it needs — **p3**.

1.3.2 methods — the right way...

In the previous section we define some Python functions that were closely associated with instances of the class **Point**: **distance_from_origin** and **_init_**. Such functions are called **methods**, and they may compute and report some property related to a class instance (that's what **distance_from_origin** does), or they may change (AKA mutate) some properties of the instance (that's what **_init_** does). What makes them methods is that they are attached to a class instance (indeed, we can call them using a reference to a class instance), and they usually use information from that instance to do their work.

Normally we don't define a function outside a class and then attach it to form a method, as we did in the previous section. Methods are so useful and common, that they are automatically defined when we have a definition indented within the scope of a class declaration. Here's what we'd do for **Point._init_**:

```
>>> class Point:
...     def __init__(self, x, y):
...         """(Point, float, float) -> NoneType
...
...         Initialize new point self with horizontal coordinate x and
...         vertical coordinate y.
...         """
...         self.x, self.y = float(x), float(y)
```

Now that we're doing things properly, our method has a docstring that includes a type contract and a brief description that mentions every parameter. If you're wondering why the type contract returns **NoneType**, it helps to know that this method takes a vanilla new object and adds attributes to it. Also, there's nothing special about the name **self** for the first parameter. We could have used **point**, **me**, **yo**, or **ego** and Python would have understood this first parameter as referring to the new instance of **Point** in the body of the **_init_** method. What counts is the position: the first parameter in a method definition is a reference to the class instance the method belongs to.

init is an example of a method that changes, or mutates, its instance. We can also define a method that computes, and then returns, some information about an instance of **Point**. In the example below, we assume that the class header for **Point** has already been define somewhere above (perhaps in the previous example...):

```
...     def distance_from_origin(self):
...         """(Point) -> float
...
...         Return the distance of self from origin.
...
...         >>> p = Point(3.0, 4.0)
...         >>> p.distance_from_origin()
...         5.0
...         """
...         return (self.x**2 + self.y**2)**(1/2)
```

Once again our properly-defined method has a docstring with a type signature and description. In addition, it has an example of how we expect the function to behave. When you write methods that return non-NoneType values, you should write one or more usage examples. These help shape your thinking when you write the body of the method, and they can provide an automatic sanity-check that your function behaves as you expect, by simply adding:

```
>>> if __name__ == '__main__':
```

```
...     import doctest
...     doctest.testmod()
...
```

...to the end of the module with your class declaration in it. Suppose I had defined **Point** in a file called **point.py**. Then evaluating the module **point** would report whether the examples matched their expected output.

Make a habit of providing docstring examples whenever your write a method or function that returns a non-NoneType value. Be sure to verify that your examples work as you expect.

special methods

Python classes inherit special methods from their common ancestor **object**. If a class developer does nothing, the methods are used as-is, which may be inappropriate. We also have the alternative of **overriding** the inherited methods, so that they have new behaviour.

You've already seen an example of a special method: **__init__**. The clue that it's a special method are those leading and trailing _s. If our declaration of **Point** had no definition of **__init__**, you could still invoke:

```
p = Point()
```

...and end up with an instance of **Point** that has no **x** or **y** coordinates. That omission would, in turn, mess up **distance_from_origin**, which expects to use **x** and **y**.

Another useful special method is **__eq__**, to check whether some object is equivalent to a given object. In the case of **Point**, two points are equivalent if they have the same **x** and **y** values, so a reasonable implementation of **__eq__** might be:

```
    def __eq__(self, other):
        """(Point, object) -> bool

        Return whether Point self is equivalent to other.

        >>> p1 = Point(3.0, 4.0)
        >>> p2 = Point(3.0, 4.0)
        >>> p3 = Point(12.0, 5.0)
        >>> p1 is p2
        False
        >>> p1 == p2
        True
        >>> p1 == p3
        False
        >>> p1 == "point"
        False
        """
        return (type(other) == type(self) and
                self.x == other.x and self.y == other.y)
```

Again, we start with the function header and a docstring. This time we have several examples, to show how that we expect two separately-declared **Point**s with the same **x** and **y** values to **not** be the same — they are different objects! However, we expect them to be **equivalent**, and the == operator is an alias for **__eq__**, and we expect it to evaluate to **True** — they are **equivalent**. Finally, we don't expect our **Point** to be equivalent to any instance of another class, for example a **str**.

The implementation returns a boolean expression that combines (with **and**) all the things we require to be true.

While you are developing a class you may need to see a **str** representation of a class instance, and even be able to evaluate that representation into an equivalent instance, so that you can examine it. The special method **__repr__** is your friend. One way to think of it is as a method that gives you a constructor for an equivalent object:

```
def __repr__(self):
    """(Point) -> str

    Return a str representing self that produces an equivalent
    Point when evaluated in Python.

    >>> p = Point(3.0, 4.0)
    >>> repr(p)
    'Point(3.0, 4.0)'
    >>> p.__repr__()
    'Point(3.0, 4.0)'
    >>> p
    Point(3.0, 4.0)
    """
    return "Point({}, {})".format(repr(self.x), repr(self.y))
```

Notice that there are several ways to call special method **__repr__**.

Sometimes you want a **str** representation of an object that easy to understand, but perhaps won't create an equivalent object in a Python interpreter. If you don't implement **__str__**, Python will use **__repr__** in its place. If you don't like that, you should implement **__str__**:

```
def __str__(self):
    """(Point) -> str

    Return a convenient str representation of self.

    >>> p = Point(3.0, 4.0)
    >>> p.__str__()
    '(3.0, 4.0)'
    >>> print(p)
    (3.0, 4.0)
    >>> str(p)
    '(3.0, 4.0)'
    """
    return "({}, {})".format(str(self.x), str(self.y))
```

Notice that there are a two ways of calling **__str__**, and that **__str__** is what's used when we **print** an instance of a class.

1.3.3 methods versus module-level functions

Class methods are very useful, but this isn't the only sort of function you should be ready to define. You already know how to define functions that are not inside any class, but simply inside a module — the **def** keyword sits at the left margin of the *.py file that defines a module.

How do you decide when to define a module-level function, and when to define a method inside a class? Ask yourself whether the function is closely associated with a class: does it change the class attributes or report on the state of the attributes in the class? If so, it is a good candidate for a method. On the other hand, if the function is less closely associated with a class, and quite reasonably takes in all the information it needs through its list of arguments, then it is a good candidate for a module-level function.

Another decision to make is whether your function should return some non-**NoneType** value (for example **str** or **int**), or is it called in order to change (mutate) some object or produce some input/output effect (for example, some **print** statement). Explicit non-**NoneType** return values are particularly straightforward to test, and you should always consider this approach. In any case, think about these issues when you write your type signature.

Some beginners mess up by expecting a function to return some non-**NoneType** value, but leaving one or more branch for **NoneType** to sneak through. An example of how this could happen is to have an **if**... without a matching **else**. One way to guard against this is to make the last statement in the function an **Exception** something like:

```
raise Exception('Should not inadvertently return None!')
```

In this case, you should write your function to have explicit **return** statements **before** reaching the **Exception**. The purpose of the **Exception** is to stop the program with a meaningful message if there is an inadvertent execution path that would return **NoneType**.

Another tricky feature of Python for some beginners involves default parameters that are of mutable (changeable) type. Keep in mind that default parameters have a value that is stored when the function is defined, and persists for each use of the function. Here is some code that illustrates this surprising feature:

```
>>> def f(n, L=[]):
...     L.append(n)
...     return L
...
>>> f.__defaults__
([],)
>>> f(3)
[3]
>>> f.__defaults__
([3],)
>>> f(4)
[3, 4]
>>> f.__defaults__
([3, 4],)
```

May Python programmers get bitten by this feature! Unless you intend this effect, you should **never** use a default parameter that can be changed (such as **list L**). A better idea is to use a special default value, such as **None**, and then test that condition to create an empty list.

1.3.4 attributes

As well as representing the behaviour of a class, using methods, we need to represent its attributes. We could simply place an assignment statement within the body of a class, like so:

```
    # [detail for class point omitted]
```

```
    z = 15

    def __init__(self, x, y):
        """(Point, float, float) -> NoneType

        [detail omitted]
        """
        # [detail omitted]
```

With this set-up, every **Point** would have a **z** with value 15. That's fine if **z** is intended to be a constant of class **Point**, although a common convention for constants is to name them with a capital letter.

If, on the other hand, we want **z** to have different values for different points, then we should set that value when we initialize the **Point**. Just add another parameter to **__init__**:

```
    def __init__(self, x, y, z):
        """(Point, float, float, float) -> NoneType

        Initialize new Point self with horizontal coordinate x,
        vertical coordinate y, and depth component z.
        """
        self.x, self.y, self.f = float(x), float(y), float(z)
```

Now we're initializing a 3-dimensional **Point**! This probably messes up **distance_from_origin** (again).

Attributes in Python are exposed (public) to any code that uses the class. This means it is completely possible for incompetent or malicious code to mess with a **Point**, for instance:

```
>>> p = Point(3.0, 4.0)
>>> p
Point(3.0, 4.0)
>>> p.x = "three"
>>> p
Point('three', 4.0)
```

Python's approach is to, at first, allow any client (code that uses our code) of our class instance to be able to both evaluate, and to change, its attributes. There is a Python **convention** (programming good manners) that any attribute that is preceded by an underline should **not** be used by client code. For example, if we had named the horizontal component of a **Point** **_x** rather than **x**, then well-behaved Pythonistas wouldn't look at, or change, **p._x** for **Point p**. It's important to note that Python doesn't **enforce** this convention — nothing stops a rude Python programmer from changing **p._x** to refer to a different value.

The advantage of this approach is that we can create classes and attributes without, initially, creating methods to control access to the attributes. This makes initial code development easier.

The downside is, as we saw above, unrestricted access to attributes can completely destroy the intended use of class instances. In many cases we'd like to filter the possible values that an attribute can refer to, or even make the attribute read-only once an instance of a class has been created. For example, it would be reasonable to make the coordinates of **p = Point(3.0, 4.0)** read-only, but still allow client code to see these values.

Python has a mechanism to allow us to change the access to an attribute after-the-fact, **without** changing the public interface our client code uses.

privacy and property

Python has no notion of attributes being private (some languages do), so they are all accessible to client code. However, Python does provide the **property** mechanism to re-direct access to an attribute, so that it must use a method.

Here's an example. In our original implementation of **Point** there was no control over access to coordinates **x** and **y**. Suppose this code was shipped, and then the developers decided that they wanted to make sure that the coordinates could not be changed after they were initialized. Since clients around the globe are already using version 0.1 of **Point**, it is not feasible to ask them to change all their code that uses **Point** — it is essential to keep the public interface the same. But we use Python's built-in **property()** function to re-direct access to the coordinates to methods that check whether the coordinate is being re-set. Here's how we do this for **x**, and the approach for **y** is completely analogous.

First, we create a method to set **x** to its initial value. This can only be done during initialization, because we check whether the name _x is already defined for the object **self**. Notice that, internally, we store this value in **_x**, using the underlined naming that Pythonistas are so polite about not tampering with.

```
def set_x(self, x):
    """(Point) -> NoneType

    Assign x to self.x
    """
    if '_x' in dir(self): # already set!
        raise Exception('Cannot change x coordinate.')
    else:
        self._x = float(x)
```

Next, we make sure that anybody wanting to use the value of attribute **x** actually uses the value referred to by **_x**:

```
def get_x(self):
    """(Point) -> float

    Return the value of coordinate x

    >>> p = Point(3.0, 4.0)
    >>> p.get_x()
    3.0
    """
    return self._x
```

None of what we've done would help unless we tell Python to re-direct all attempts to assign to, or evaluate, **x** to the appropriate methods. We use function **property()** which takes four arguments: a method to get the value of **x**, a method to set the value, a method to delete attribute **x**, and some documentation. The arguments may, optionally, be **None**, to indicate methods we haven't implemented:

```
x = property(get_x, set_x, None, None)
```

And that's all there is to it. All the client code that evaluates or assigns to **x** is now redirected to **get_x** and **set_x**. Even the code we have written in the methods of class **Point** itself (for example in **__init__**) gets redirected!

1.3.5 composition

It is a bit humbling to realize that a user-defined class, such as **Point**, is now a fully-fledged type, and equal with types such as **int** or **str**.

That means you can use user-defined types in Python code anywhere you would use built-in types. That includes using user-defined types **inside** new user-defined types.

This approach is called **composition**. If we need the features of type **A** in a user-defined type **B**, we simply include an instance of **A** in our declaration of class **B**.

Our definition of class **Point** included instances of class **Float** — the coordinates **x** and **y**.

We could define a new class, **Triangle**, that uses instances of **Point** to represent the three corners (vertices):

```
class Triangle:
# ... stuff omitted
   def __init__(self, c1, c2, c3):
      """ (Triangle, Point, Point, Point) -> NoneType

         Create a new Triangle self with corners c1, c2,
         and c3.
      """
      (self.corner1, self.corner2, self.corner3) = (c1, c2, c3)
      # ... lots more stuff omitted
```

Now we concentrate on writing methods and creating attributes for class **Triangle**, without re-creating any of the methods and attributes of class **Point**. Any time we need **Point** data or behaviour, we get it seamlessly from class **Point**, using instances **self.corner1**, **self.corner2**, and **self.corner3**.

Composition is a very powerful approach. We build increasingly complex types using already-built types, thus reducing the amount of complexity we have to deal with at any given time. With **Triangle** implemented, we can consider new classes based on it, perhaps **Pyramind** or even **Bermuda**. We focus on the design and implementation of each new class, while we depend on, and use, the work already done on the classes we incorporate into it.

1.4 inherit features

Sometimes we create a new class that adds a few features to an existing class, but still represents the same sort of thing. In order to save code, we could always include an instance of the existing class in our new class, and use its features. This approach is called **composition**. However, if we decided that our new class really **is** a special case of the existing class, we can create a **subclass** that, by default, inherits all the methods and attributes of the class we declare to be its parent.

We do this in the class declaration by adding the name of one or more existing classes in parentheses:

```
from point import Point


class HeavyPoint(Point):
# ... lots of stuff omitted
```

Recall the **Point** class from earlier. If we want to specialize to a **Point** that also has a mass, it doesn't make sense to re-write the code from **point.py**. That's tedious, and tedium breeds errors.

Inheritance allows us to **inherit** some of the methods of **Point**, to use and modify, or **extend** some of its methods, and to completely replace, or **override** other methods. Here are some examples.

HeavyPoint._str_ needs to add a factor of **f** to the **str** representation of **self**. Notice how it re-uses the alread-existing code by calling **Point._str_**:

```
def __str__(self):
    """(HeavyPoint) -> str

    Return a convenient str representation of self.

    >>> p = HeavyPoint(1.0, 2.0, 3.0)
    >>> print(p)
    3.0(1.0, 2.0)
    """
    return '{}{}'.format(str(self.m), Point.__str__(self))
```

This approach is called **extending** the parent class.

HeavyPoint._add_ needs to produce a new **HeavyPoint** using the sums of all three parts: the **x**, **y** and **m** part. I don't see an easy way to re-cycle the existing code, so I **override** it — create a completely new method with the same name:

```
def __add__(self, other):
    """(HeavyPoint, HeavyPoint) -> HeavyPoint

    Return component-by-component sum of self and other.

    >>> hp1 = HeavyPoint(1.0, 2.0, 3.0)
    >>> hp2 = HeavyPoint(4.0, 5.0, 6.0)
    >>> hp1 + hp2
    HeavyPoint(5.0, 7.0, 9.0)
    """
    return HeavyPoint(self.x + other.x, self.y + other.y,
                      self.m + other.m)
```

Some of the methods from **Point** work without any modification in **HeavyPoint**: **distance_from_origin** and **scale**, for example. Without writing any code in **HeavyPoint**, we get these for free.

How does Python decide which version of a method to use when it might be defined in a class, its superclass (or the superclass of the superclass...)? It checks first in the most specific class that has been declared. If it doesn't find the method defined there, it checks the superclass (there may be more than one). If it doesn't find it there, it checks the superclass's superclass. And so on. The first method it finds is the one it uses.

Using this approach, Python uses the **_repr_** method defined in **HeavyPoint** and ignores the one defined in **Point**. We have a mixture of approaches for **HeavyPoint._eq_**, since Python finds a definition in **HeavyPoint**, but this definition includes a reference to **Point._eq_**. This means it uses the definition from **HeavyPoint**, which in turn uses the definition from **Point**.

In order to challenge your understanding of inheritance, read over the bare-bones class declarations below, and predict the output of the two **print** statements **before** running the code. Explain your results.

```
class A:

    def g(self, n):
        return n
```

```
    def f(self, n):
        return self.g(n)

class B(A):

    def g(self, n):
        return 2*n

if __name__ == '__main__':
    a = A()
    b = B()
    print("a.f(1): {}".format(a.f(1)))
    print("b.f(1): {}".format(b.f(1)))
```

1.5 raise exceptions

Programmers usually plan for ordinary circumstances. For example, **IntStack.pop** is designed to return an integer. Under exceptional circumstances, if this is called on an empty **IntStack**, it doesn't return any value: it raises a **PopEmptyStackException**. If the client code, which called **IntStack.pop** knows what to do with a **PopEmptyStackException**, it does, otherwise it also stops running and passes it on to what ever code called it. If no part of the program knows what to do, execution stops with information about what happened:

```
>>> i.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/heap/148W15/Notes/int_stack.py", line 53, in pop
    raise PopEmptyStackException
int_stack.PopEmptyStackException
```

Without defining a special **Exception**, every programmer has (usually inadvertently) triggered **Exception**s of their own. Try out these expressions:

```
>>> int("seven")
>>> a = 1/0
>>> [1, 2][2]
```

Each expression raises some sort of variety, and if the client code that calls it (in this case, the prompt at the Python interpreter) doesn't know what to do, it passes it on until the program eventually stops ungraciously or "crashes."

You can also deliberately raise an **Exception** of some variety that has already been defined in Python. Try out:

```
>>> raise ValueError
>>> # or...
>>> raise ValueError("that value is wrong in so many ways!")
```

What is the advantage of defining and raising **Exception**s, rather than just writing code to deal with exceptional circumstances in the same part of the program where they might occur? A problem with that approach is that exceptional circumstances may occur in many places (dividing by zero might occur

almost anywhere), and it would take a lot of repetitive code to deal with them in every piece of potentially problematic code. Python uses a pair of **try... except** blocks to detect and deal with those **Exception**s that the programmer knows how to deal with. We might deal with the possibility of an empty **IntStack** by letting the user know what's up:

```
try:
    n = i.pop()
except PopEmptyStackException:
    print("You cannot get an integer from an empty stack!")
```

In this case, rather than crash, the program prints some information and continues. The **try** block can contain multiple lines of code, and in our example, any code that raised a **PopEmptyStackException** — even indirectly — would be dealt with.

Exceptions are classes, all descended from class **Exception**, even if they have an empty body:

```
class PopEmptyStackException(Exception):
    """Raised when pop from empty stack attempted"""
    pass
```

This means that a **PopEmptyStackException** is also an **Exception**, so any **try... except** blocks that deal with vanilla **Exception**s will also deal with **PopEmptyStackException**s the same way. You can even design a hierarchy of **Exception** subclasses and sub-subclasses. Then your **try... except** blocks can be specialized to first try to deal with the most specific **Exception**s, then more general **Exception**s, and then deal with all the **Exception**s that weren't covered by any other block. Here's an example:

```
class SpecialException(Exception):
    pass

class ExtremeException(SpecialException):
    pass

if __name__ == '__main__':
    try:
        #1/0
        #raise SpecialException('I am a SpecialException')
        #raise Exception('I am an Exception')
        raise ExtremeException('I am an ExtremeException')
        1/0

    # use name se if a SpecialException detected
    except SpecialException as se:
        print(se)
        print('caught as SpecialException')
    # use name ee if ExtremeException detected
    except ExtremeException as ee:
        print(ee)
        print('caught as ExtremeException')
    # name all other Exceptions e
    except Exception as e:
        print(e)
        print('caught as Exception')
```

You should copy and experiment with this code. Try removing and inserting the # character on various lines in the **try** block. Explain the result. Is there any expression you can put in the **try** block that will cause an **ExtremeException** to be detected in this code? Why or why not?

Chapter 2

development

2.1 collaborate productively

Code development doesn't happen in isolation: programmers usually work in teams, and they often look at similar problems from books, on the web, etc. Good collaboration should lead to better understanding of your code. Bad collaboration leads to thoughtless copying and potentially even plagiarism with all its consequences.

Sometimes students collaborate by dividing a programming assignment up into roughly equal parts, working on the parts separately, and then combining them shortly before they are submitted. This approach (bad collaboration) almost guarantees that each student will be ignorant of the solutions crafted by her/his peers.

A better approach is to combine division of labour with code review. Each team member should review, understand, and try to improve the work of their peers. Learn (although, unfortunately, not in this course) how to use a revision control system such as **git** or **subversion** to allow shared revisions of the same source files. This allows you to roll your code back to an earlier version, when needed, or identify which team member added a particularly brilliant feature...

A very productive practice is to have a different team member from the one who was the author of a class or module write tests for it. A fresh pair of eyes (and fingers on the keyboard) may turn up flaws that were, effectively, invisible to the original coder.

Don't copy code solutions that you see online.[1] As a mature developer, you should be able to consider the strengths, weaknesses, and main ideas, in somebody else's solution as you develop your own.

2.2 re-use

As a programmer you should be on the look-out for duplication of code. Duplicate code is not only extra work, but it is a constant source of error: any time the same code exists in two places, there is a danger (a near-certainty) that it will be updated in one place and not the other.

A good way to avoid duplication is to re-use existing code, whether it's your own function or method, or part of your programming language's standard toolkit. Here are some examples of Python functions that you should rely on rather than duplicate (you should be sure to read Python documentation on these, in order to make the best use of them):

[1] But do give credit to any source you have consulted for ideas.

2.2.1 sum

Given a list of numerical values, all of us can figure out how to write a few lines to sum those values. However, if we're hurrying, we might forget to set the initial sum to 0, so that the sum of an empty list computes correctly. This a common enough task that it is built-in to Python:

```
>>> sum([5, 4, 3, 2, 1])
15
```

2.2.2 min and max

Finding the minimum of a list of values isn't too hard, but you do need to make sure that you choose a candidate for the minimum that is as big as at least one list element. Once again, rather than re-invent the wheel and possibly make errors, use what's built-in:

```
>>> min([7, 5, 3, 8])
3
```

There's a corresponding maximum function. These work on types other than numbers, indeed any types that support greater than and lesser than:

```
>>> min(['how', 'now', 'brown', 'cow'])
'brown'
```

Why is 'brown' the least string in that list? A good way to think about this is to consider how you compare strings, for example words in a long list.

```
>>> min([(7, 5), (5, 3), (3, 5), (3, 7)])
(3, 5)
```

Why is the tuple **(3, 5)** the least in that list? You could experiment with comparing tuples with the $<$ operator.

2.2.3 list comprehension

If you wanted to produce a sublist of **L**, from index 3 to 16, you could roll your own code:

```
new_list = []
for i in range(len(L)):
   if 3 <= i < 17:
      new_list.append(L[i])
```

However, you would be likely to use Python's **slice** expression instead, since it is short, clear, and reliable:

```
L[3:17]
```

Similarly, if you wanted to produce a new list consisting of the square of each element in **L**, you could write your own:

```
new_list = []
for n in L:
   new_list.append(n**2)
```

However, Python provides a **list comprehension** expression that is also short, clear, and reliable:

```
[n**2 for n in L]
```

List comprehensions make it clear that you are producing a new list and not modifying the old list **L**. Like slice, they are expressions and can occur anywhere an expression can, for example they can be **return**ed.

2.2.4 set

Sometimes you want to be sure that a collection of objects has no duplicates. Provided the objects are **immutable**, you can make the collection into a **set**, which ensures there is exactly one instance of each element:

```
>>> L = [1, 2, 3, 2, 5]
>>> set(L)
{1, 2, 3, 5}
>>> L = list(set(L))
>>> L
[1, 2, 3, 5]
```

Sets also support operations such as set union, intersection, and difference:

```
>>> A = {1, 2, 3}
>>> B = {3, 4, 5}
>>> A | B
{1, 2, 3, 4, 5}
>>> A & B
{3}
>>> A - B
{1, 2}
```

2.2.5 any, all

If you wanted to write a function that determined whether at least one element of list **L** evaluate to **True** in a boolean context, you could write a few lines of code:

```
>>> def any_true(L)
>>>     for x in L:
>>>         if x:
>>>             return True
>>>     return False
```

This is a common enough situation that Python provides a built-in function **any**:

```
>>> any(L)
```

It's worth noting that most Python values evaluate to either **True** or **False** in a boolean context, so you should be ready for surprises. What do you predict in these cases:

```
>>> any([3, 0, 1])
```

. . . or

```
>>> any([0, []])
```

The flip side of **any** is **all**. You could write your own function to determine whether all elements of **L** evaluate to **True** in a boolean context:

```
>>> def all_true(L)
>>>     for x in L:
>>>         if not x:
>>>             return False
>>>     return True
```

. . . or use the built-in function **all**:

```
>>> all(L)
```

Again, be aware that most Python values evaluate to either **True** or **False** in a boolean context. Before typing them into a Python interpreter, try to predict what is produced by these expressions, and explain your prediction.

```
>>> all([1, True])
>>> all([True, False])
>>> all([])
```

2.2.6 conditional expression

Sometimes it is handy to have a condition that evaluates as an expression. Python's familiar **if**... **then**... blocks cannot, for example, be **return**ed by a function (try it!) since they are not expressions. However, Python does provide a conditional expression:

```
>>> a = 5
>>> 12 if a > 6 else 11
11
>>> a = 7
>>> 12 if a > 6 else 11
12
```

The expression has three parts, and all three must be present:

1. the true part, before the keyword **if**;
2. the condition, between keywords if and **then**;
3. the false part, after the keyword **then**.

If the condition is **True**, the entire expression evaluates to the true part. Otherwise, it evaluates to the false part.

2.3 break problems down stepwise

We want our functions and methods to solve problems in a clear way — clear enough for a computer to execute, and clear enough for a human to understand, modify, and maintain.

However, many of the problems are complex, and that often means their solutions are complex. One approach to maintain clarity is to break our solution down into simpler steps that we can combine. Since it may be difficult to begin by thinking of suitably simple intial steps that can be combined into a solution, we often work the other way around: assume you have a solution and repeatedly refine it into simpler and simpler steps. This is often called "top-down design" or "step-wise refinement."

Here's an example. A completed Sudoku puzzle is an **n**×**n** grid of symbols where:

1. **n** is a square integer, such as 1, 4, 9, 16, ...
2. Every row, column, and $\sqrt{n} \times \sqrt{n}$ subsquare contains exactly one of each symbol from a set of **n** symbols, for example one each from $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ if **n=9**.

Heres a **9**×**9** grid of numbers:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Sudoku example from
https://upload.wikimedia.org/wikipedia/commons/thumb/3/31/Sudoku-by-L2G-20050714_solution.svg/
364px-Sudoku-by-L2G-20050714_solution.svg.png

If you have a wonderful visual cortex, you can somehow glance at this and instantly verify that it satisfies condition 2 above. Otherwise, provided you know that the **DIGIT_SET** is $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, you need to do something like this:

```
def valid_sudoku(grid, digit_set):
    """
    Return whether grid represents a valid,
    complete sudoku.

    @type grid: list[list]
    @type digit_set: set
    @rtype: bool

    Assume grid is square (as many rows as columns)
    and has the same number of rows as elements of
    digit_set.

    >>> valid_sudoku(_GRID, _DIGIT_SET)
    True
    >>> g = [[x for x in row] for row in _GRID]
    >>> g[0][1] = 5
    >>> valid_sudoku(g, _DIGIT_SET)
    False
    """
    assert all([len(r) == len(grid) for r in grid])
    assert len(grid) == len(digit_set)
    return (_all_rows_valid(grid, digit_set) and
            _all_columns_valid(grid, digit_set) and
            _all_subsquares_valid(grid, digit_set))
```

You first reaction should be "that doesn't look like a solution at all, just new problems!" However, with luck and insight, the new problems should be simpler than the original. Since the three subproblems have a similar flavour, let's attack the first one, also in a top-down way:

```
def _all_rows_valid(grid, digit_set):
    """
    Return whether all rows in grid are valid and complete.
```

```
    @type grid: list[list]
    @type digit_set: set
    @rtype: bool

    Assume grid has same number of rows as elements of digit_set
    and grid has same number of columns as rows.

    >>> _all_rows_valid(_GRID, _DIGIT_SET)
    True
    >>> g = [[x for x in r] for r in _GRID]
    >>> g[0][1] = 5
    >>> _all_rows_valid(g, _DIGIT_SET)
    False
    """
    assert all([len(r) == len(grid) for r in grid])
    assert len(grid) == len(digit_set)
    return all([_list_valid(r, digit_set) for r in grid])
```

Again, a first reaction might be "we haven't made any progress here!" But, maybe we have. A valid row is one that has all the digits in $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ exactly once. In other words, we want something like:

```
def _list_valid(r, digit_set):
    """
    Return whether r contains each element of digit_set exactly
    once.

    @type r: list
    @type digit_set: set
    @rtype: bool

    Assume r has same number of elements as digit_set.

    >>> _list_valid(_GRID[0], _DIGIT_SET)
    True
    >>> L = [x for x in _GRID[0]]
    >>> L[1] = 5
    >>> _list_valid(L, _DIGIT_SET)
    False
    """
    assert len(r) == len(digit_set)
    return set(r) == digit_set
```

. . . in Python, **set(r)** produces a set from a list of immutable elements.

Part of our step-wise refinement is done. You might (and perhaps should) react with indignation that extracting the rows from **grid** is deceptively easy, and extracting columns or subsquares will be harder, and you'd be justified. However, the spirit of step-wise refinement is to make harder problems easier by breaking them, so we could use this approach with columns:

```
def _columns(grid):
```

```
    """
    Return list of columns in 2D list grid.

    @type grid: list[list]
    @rtype: list[list]

    Assume grid is square, i.e. has same number of rows and
    columns.

    >>> G = [[1, 2], [3, 4]]
    >>> _columns(G)
    [[1, 3], [2, 4]]
    """
    assert all([len(r) == len(grid) for r in grid])
    return [_column(i, grid) for i in range(len(grid))]
```

Again, we answered a question with a question: "how do you produce a list of columns in **grid**" is replaced with "how do you produce the **ith** column in **grid**?" Perhaps the second question is easier to answer.

```
def _column(i, grid):
    """
    Return the ith column of 2D list grid

    @type i: int
    @type grid: list[list]
    @rtype: list

    Assume i is a valid index for a column of grid, and that
    grid has same number of rows and columns.

    >>> G = [[1, 2], [3, 4]]
    >>> _column(1, G)
    [2, 4]
    """
    assert all([len(r) == len(grid) for r in grid])
    assert i in range(len(grid))
    return [grid[j][i] for j in range(len(grid))]
```

You should notice that, as we refine the problem, we write the new solutions as helper functions (or methods). This helps focus our mind on just a small piece of a big problem. It also means that we can fit most of the code we are working on at any given time onto our screen (and into our brain). Finally, it means we can test the helper function to increase our confidence that it does what we meant it to do before we use it to... help.

You're welcome to look at, and tinker with, the rest of the code for sudoku_code.

2.4 document

You tell your clients what your functions and classes do (their public interface) through **docstrings** (see page 3). Once you are used to the **function design recipe**, you'll likely spend most of your time writing (and

re-writing) your docstring. Then, after you have carefully understood the purpose and structure of, say, a new function, the body of the function is relatively short and simple to write. Get in the habit of writing a complete docstring before you write the body of a function — it will guide you. Also, if you're seeking advise from a TA or instructor on your code, we'll expect to see a docstring.

If you are using an IDE that supports type-hinting (e.g. **PyCharm**), you should modify the **function design recipe** so that the signature can be checked automatically whenever the function is referred to inside the IDE:

```
def is_odd(num):
    """
    Return whether num is 1 more than some multiple of 2.

    @type num: int
    @rtype: bool

    >>> is_odd(5)
    True
    >>> is_odd(0)
    False
    """
    return num % 2 == 1
```

Now, your IDE should warn you if you try something like this:

```
is_odd("five")
```

. . . or this:

```
3 + is_odd(7)
```

You will also need to document classes you write. The class design recipe outlines a step-by-step procedure for declaring a class, its methods and attributes. Any clients of your class will need this documentation to use your software, and instructors/TAs will need to see this to provide help at office hour.

2.5 test

2.6 refactor

2.7 debug

Chapter 3

recursion

Recursion is a useful and beautiful problem-solving technique. Every time we can break a problem into smaller problems with the same structure, solve them, and combine the solution into a solution to the original problem, we have used recursion.

Most programming languages provide some support for recursion.

3.1 names reduce repetition

As a programmer, you would never dream of typing and re-typing the literal value of an approximation of the ratio between the circumference and diameter of a circle, π:

$$3.141592653589793... \tag{3.1}$$

.

We would either name a constant:

```
PI = 3.141592653589793...
```

. . . or, better yet, use the name of a constant that developers of standard modules have already created:

```
import math
math.pi
```

Similarly, if you were implementing a function that required the length of a list, you would either name and implement a helper function that performed this task, or (even better) use the built-in **len**:

```
def square_list(L):
    ''' (list) -> list

    Produce a new list with L's elements
    repeated len(L) times.
    '''
    return L * len(L)
```

This is the insight that recursion continues. If you are in the middle of implementing a function that adds all the numbers in an arbitrarily-nested list of numbers, and you find that you need to add all the numbers in an arbitrarily-nested sublist of numbers, you don't re-invent the wheel (or the function).

You simply use the **name** of the function that you are in the middle of implementing to direct program execution to the solution to the sub-problem (or sublist, in this case):

```
def sum_list(L):
    ''' (list) -> int

    Return sum of numbers in L or in L's possibly-nested sublists.

    Assume L is a list of numbers or, possibly, other lists of the
    same structure.

    >>> sum_list(17)
    17
    >>> sum_list([1, 2, 3])
    6
    >>> sum_list([1, [2, 3, 4], 5])
    15
    '''
    # reuse: isinstance, sum, sum_list !
    if not any([isinstance(x, list) for x in L])
        return sum(L)
    else: # L contains at least one sublist
        return sum([sum_list(x) if isinstance(x, list)
                    else x for x in L])
```

The above implementation names built-in functions **isinstance** and **sum** as helper functions. It also uses **sum_list** — the function that is being implemented! In the next section we trace how this self-reference can yield the solution we need.

Function **sum_list** will sum numeric elements of lists no matter how deeply and complexly nested the lists are. We say that a simple list such as **[1, 2, 3]** has **depth 1**. In general, we say that a list has depth 1 more than the depth of its most deeply-nested list element (non-lists have depth 0). So:

- `[1, [2, 3, 4], 5]` has depth 2;
- `[6, [1, [2, 3, 4], 5], 7, [1, [2, 3, 4], 5]]` has depth 3;

. . . and so on.

3.2 follow recursion

To understand the workings of a recursive function such as **sum_list**, you should trace examples that work up from simple to more complex. Each time you see a recursive function call that is equivalent to something you have already traced **stop tracing** and fill in the value directly. Here are examples:

```
sum_list([1, 2, 3]) -> sum([1, 2, 3]) -> 6
# sum_list returns the sum of the values of depth 1 arguments
sum_list([1, [2, 3, 4], 5])
-> sum([1 , sum_list([2, 3, 4]), 5])
-> sum([1, 9, 5]) -> 15
# sum_list returns the sum of the values of depth 2 arguments
```

It is **very important** in the example above that we didn't trace `sum_list([2, 3, 4])` any further. We simply replaced it by the value we knew it must have, since we had **already seen (and written a comment)** that with a

list of depth 1 **sum_list** returns the sum of the values. We don't re-trace recursive calls we understand, since the resulting explosion of recursive calls would obscure our understanding of recursion.

3.3 design recursion

Once you are able to follow somebody else's recursive function definition and verify that it satisfies the docstring, it's time to implement some recursive functions of your own. In the examples that follow, your job will be to make your design work for two different cases:

Case 1, base case: If you cannot decompose the problem your function is solving into smaller problems of the same sort, you have a base case. That means you solve the problem and (likely) return a result without any recursion — in other words directly.

Case 2, general case: If the problem your function is solving can be decomposed into smaller problems of the same sort, and the solution of those smaller problems allows you to solve the original problem, you have a general case. You should solve the problem by recursively calling your function on the smaller problems, and then combining the results of these recursive calls.

This will sound less vague when we look at some examples.

3.3.1 lists of lists

A Python **list** may contain other Python **list**s — it is a recursive data structure. If we define a Python **list** with no sublists has having depth 1, and a Python **list** with one or more sublists has having depth 2, and so on, the question comes up of how to write a program to find the depth of a Python **list**. Start with the function design recipe:

```
def list_depth(L):
    ''' (list) -> int

    Return the depth of L, a possibly-nested
    Python list.

    >>> list_depth([])
    1
    >>> list_depth([1, 2, 3])
    1
    >>> list_depth([1, [2, 3], 4])
    2
    '''
```

Looking over the examples, it seems clear that the base cases are lists that do not contain any sublists. We can detect such lists by checking for any sublists:

```
    if not any([isinstance(x, list) for x in L]):
```

Such lists (sometimes called "flat" lists) have depth 1, so we can implement that body of the **if** branch:

```
    if not any([isinstance(x, list) for x in L]):
        return 1
```

The other case is when **L** contains one or more sublists. We need to find the maximum depth of these sublists, and then add 1 to account for the extra depth added by **L** itself. Since we have a built-in **max** function, we'd like to generate a list of the depths of **L**'s sublists. One way to do this is with a list comprehension, but we'll need to make sure that any non-list elements of **L** do not contribute to the maximum. Here are a couple of approaches:

```
if not any([isinstance(x, list) for x in L]):
    return 1
else:
    return 1 + max([list_depth(x) for x in L if isinstance(x, list)])
```

The clause **if isinstance(x, list)** acts as a filter on the resulting list comprehension — only elements that satisfy this condition are considered, and then they have **list_depth** evaluated on them. Another approach would be to note that, since there are sublists of depth at least 1, any zeros in the resulting list will have no effect on the maximum:

```
if not any([isinstance(x, list) for x in L]):
    return 1
else:
    return 1 + max([list_depth(x) if isinstance(x, list)
                    else 0 for x in L])
```

3.3.2 trees

Some rich collections of information are loosely modelled on biological trees. Family trees (so long as you consider just one parent for each group of siblings) and phylogenetic trees are examples. This recursive organization turns up commonly enough that it is codified in mathematics with a set of definitions:

- set of **nodes** (possibly with values or labels), with directed **edges** between some pairs of nodes
- One node is distinguished as **root**
- Each non-root node has exactly one parent.
- A **path** is a sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_i to n_{i+1}. The **length** of a path is the number of edges in it
- There is a unique path from the root to each node. In the case of the root itself this is just n_1, if the root is node n_1.
- There are no **cycles** — no paths that form loops.
- **leaf:** node with no children
- **internal node:** node with one or more children
- **subtree:** tree formed by any tree node together with its descendants and the edges leading to them.
- **height:** 1 + the maximum path length in a tree. A node also has a height, which is 1 + the maximum path length of the tree rooted at that node[1]
- **depth:** Height of the entire tree minus the height of a node is the depth of the node.

[1] Some texts define the height as the maximum path length in a tree. Be sure you check which definition you're using, since they differ by 1. The other definition leads to the awkward (I think) situation where the height of the empty tree is -1

- **arity, branching factor**: maximum number of children for any node in the tree

Here is a Python declaration of a Tree class. We blur the distinction between a **Tree** and the **root node of a Tree**, since in code we access a tree through its root node, and we're almost always interested in its subtrees, which are also accessed through their root nodes. When the distinction makes a difference we can always translate back from "the tree" to "the tree rooted at the node..."

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    """

    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        @param Tree self: this tree
        @param object value: value contained in this tree
        @param list[Tree] children: possibly-empty list of children
        @rtype: None
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []
```

Our declaration has no way of representing an empty tree, if we needed one we could use some special value such as None to stand for it. The declaration does use **None** as a label or value, but that allows us to represent nodes without a value or label — not the same thing as an empty tree with no nodes.

Almost every question about a tree has a recursive answer. For example, for any tree except a tree where the root is also a leaf, the leaves of the tree will be identical to the leaves of the subtrees. So, in order to count the leaves, we first consider the case that involves no recursion: how do you count the number of leaves in a single-node tree, and how do you detect a single-node tree? If you count 1 leaf, and check that you have a single-node tree by noticing that it has no children, then we agree.

Next, any tree with two or more nodes has subtrees rooted at the children of the root. If we count the leaves of the subtrees, we have counted the trees leaves.

Translating from English to recursive Python, **t** is a **Tree**, then to count **t**'s leaves we define the function **leaf_count(t)**:

```
if not t.children:
    # t is a leaf
    return 1
else:
    # t is an internal node
    return sum([leaf_count(c) for c in t.children])
```

Ask a different question: what is the height of tree **t**, where we use the definition that the height is 1 plus the maximum path length from the root to a leaf. It is a single step to transform this into 1 plus the maximum height of **t**'s subtrees, and the code to implement **height(t)** almost writes itself from the definition:

```
if not t.children:
    # t is a leaf
```

```
        return 1
    else:
        # t is an internal node
        return 1 + max([height(c) for c in t.children])
```

The recursive approach to trees isn't limited to producing numbers. Another line of questioning might require answers comprised of lists of tree values. To do this recursively means to produce simple lists — for example, single-element lists — for simple trees, and then combining lists for more complex trees. To combine aggregate numerical results we used functions such as **sum** or **max**. In the case of lists, we often want to combine a list of lists into a single list. One way to do this is with a variant of the **sum** function:

```
>>> sum([[1, 2, 3], [4, 5]], [])
[1, 2, 3, 4, 5]
```

The second argument, [], tells **sum** what to do with an empty list of lists.

Apply this to the question: what is the list of all values in a tree? Our approach is to first consider the simple, non-recursive, case of a tree: a single leaf. A list of this leaf's values is easy to produce. Next consider what to do with a non-leaf (an internal node). Here produce a list of the current node's value concatenated with the lists of values in from all its children. Putting these together, an implementation of **list_all(t)** would be:

```
return [t.value] + sum([list_all(c) for c in t.children], [])
```

3.3.3 binary trees

3.3.4 binary search trees

3.3.5 linked lists

3.4 flavours of recursion

3.4.1 structural recursion

Most of the recursive functions we've written take an argument which is itself a recursive data structure (such as a tree). Since a recursive data structure includes substructure(s) of the same type, a natural approach to writing a recursive function on them is:

1. If the instance of the data structure cannot be broken down into smaller substructures of the same sort, and a solution derived by solving those smaller problems, process the data structure directly, without recursion. This is a base case.

2. If the instance of the data structure can be broken down into substructures of the same sort, with solutions for the smaller problems yielding solutions to the original problem, then process the substructures and combine the results into the result for the main structure.

This textual description allows you to focus on how to solve a base case, and how to solve a general case. If you name those two "how to solve" problems as functions **start_from** and **solve_combine**, you can write a generic function for structural recursion as:

```
def structurally_recurse_on(rds, is_base_case, start_from, solve_combine):
    ''' (object, function, function, function) -> object
```

```
    Recurse on recursive data structure rds to produce result,
    using start_from when is_base_case(rds) and combine for general case.
    '''
    if is_base_case(rds):
        return start_from(rds)
    else:
        return solve_combine(rds)
```

Whether you use this generic code, or simply think of it as a guide in writing your own code, you now focus on three problems:

is_base_case: Decide whether an instance is a base case or not.

start_from: Compute a result directly (no recursion) from a base case.

solve_combine: Recursively solve the problem for substructures, and combine those results into a solution for the main structure.

3.4.2 generative recursion

Some problems decompose into smaller problems of the same type (so a recursive solution should be investigated) by the data doesn't decompose into parts corresponding to the smaller problems. For these problems the developer needs to think about how to **generate** smaller instances of the input data structure that correspond to the smaller subproblems. This requires some extra ingenuity from the developer, and some extra caution: they must be sure that the decomposed data structures are really smaller, so that recursion will eventually terminate.

quick sort

Suppose you have a list of numbers, and you want to produce a new list with the same values rearranged into ascending order. Here's one way to do it:

1. List all elements that are smaller than the first element.
2. Make a list consisting of just the first element.
3. List all elements that are not smaller than the first element, other than the first element itself.
4. Sort the first and third list, then concatenate list 1, list 2, and list 3.

How do you sort the first and third lists? Use this same 4-step procedure recursively! But be careful: if you try to carry out these steps on an empty list you will not be able to find a first element, and you will not be able to reduce the size of the lists when you carry out recursion. How do you directly (no recursion) sort an empty list?

For a non-empty list, notice that the lists you **generate** in the first and third steps are not recursive sublists of the list you begin with. You, the developer, must write code to generate these two lists, using insight into the problem. In the case of this sorting algorithm, the insight is that the first element of a non-empty list should be placed between the list of elements smaller than it, and the list of elements no smaller than it, in a sorted variant of the same list elements. Here's the quicksort algorithm written in Python:

```
def quick_sort(L):
    ''' (list) -> list
```

```
    Return a new list with the same elements as L in
    ascending order.

    >>> L = []
    >>> quick_sort(L)
    []
    >>> L = [3, 1, 2]
    >>> quick_sort(L)
    [1, 2, 3]
    '''
    if L == []:
        # copy of L
        return []
    else:
        return (quick_sort([i for i in L if i < L[0]]) +
                [L[0]] +
                quick_sort([i for i in L[1:] if i >= L[0]]))
```

One possible problem is: what happens when the first element is the smallest, or largest, element in the list? Although the algorithm remains correct, you should be able to work out why it will be inefficient. One way to (mostly) avoid this inefficiency is to pick a random element of the list as the "pivot," rather than always choosing the first element. Try to modify the code given to use this approach.

gcd

The greatest common divisor (gcd) of two non-negative integers is the largest integer that divides them both without a remainder, for example **gcd(4, 6)=2**. If you are given two non-negative integers **n** and **m**, you can certainly calculate **gcd(m, n)** by comparing all the integer divisors of **n** and **m**, and finding the largest one. You are guaranteed at least one positive integer (which one?) will be on both lists.

Over 2000 years ago Euclid found a much more efficient method of calculating **gcd(m, n)**. He showed that **gcd(m, n)** was the same as **gcd(n, m%n)**. Of course **m%n** is not defined when **n=0**, but in this case it's easy to see that any non-zero **m** divides both **m** and **0**, so we define **gcd(0, m)=m**, even when **m** is **0**. Symmetrically, we define **gcd(n, 0)=n**.

What does all this have to do with recursion? Well, if **m**is no smaller than non-zero **n**, you can be sure that **m%n** is less than both of them. So **gcd(n, m%n)** has a smaller pair of numbers as arguments than the equivalent **gcd(m, n)**. By repeatedly reducing the pair of numbers, we should end up with a pair that includes a zero, in which case ... see the previous paragraph! Here's a Python implementation of **gcd**:

```
def gcd(m, n):
    """
    (int, int) -> int

    Return the greatest common denominator of natural numbers
    m and n.
    """
    # if either argument is 0, return the other one
    if m == 0:
        return n
    elif n == 0:
```

```
        return m
    # either m >= n OR n >= m%n
    else:
        return gcd(n, m%n)
```

Although my argument above assumes that **m**≥**n**, notice that even if **m**<**n**, the next recursive call has **n**≥**m%n**, so all's well.

Why is this **generative** recursion? Even if you constructed integers as a recursive data structure, it is very unlikely that the structure for **m** or **n** would always have a substructure **m%n**. This is another case where the developer thought hard about the problem and **generated** the data for the recursive step: **n** and **m%n**.