# PREPARING FOR THE FINAL EXAM

CS 1110: FALL 2012

This handout explains what you have to know for the final exam. Most of the exam will include topics from the previous two prelims. We have uploaded the solutions to each of these exams into CMS. Just click on the link for each exam.

In addition, the final will cover multidimensional lists and loop invariants, which you should now be familiar with after Assignment 6. The most unusual new question on the final exam will be the *required algorithms*. There is a list of algorithms that we are expecting you to know and reproduce on this final. They are listed in Section 3 below.

The final will *not* cover callback functions or GUI design. Those were useful topics for Assignment A7, but will not appear on the final. A more detailed list of exam questions is provided below in Section 2.

## 1. Exam Information

The exam will be held **Friday, December 7th from 9:00-11:30 am**. Again, for reasons of space, we are split across two rooms.

- Students with last names A – U meet in Statler 185 (the big auditorium)
- Students with last names V – Z meet in Statler 196

**Review Sessions.** Unlike the prelims, there will be multiple review sessions for this final. There is a total of nine review sessions, each lasting one hour. You are free to attend as few or as many as you wish. All of the review sessions will be held in **Hollister B14**.

The review session topics (and the TA, consultants running them) are as follows:

**Sunday, December 2nd**

- 2 – 3pm: Call Frames and Diagramming Objects (Jeran Fox)
- 3 – 4pm: Classes and Subclasses (Jeran Fox)
- 4 – 5pm: Exceptions and Try-Except (Natalie Kim)

**Monday, December 3rd**

- 1 – 2pm: Lists and Sequences (Amy Frankhouser)
- 2 – 3pm: Recursion (Caleb Perkins)
- 3 – 4pm: Open Question Session (Walker White)

**Tuesday, December 4th**

- 1 – 2pm: Loop Invariants (Caleb Perkins)
- 2 – 3pm: The Required Algorithms (Walker White)
- 3 – 4pm: Open Question Session (Walker White)

## 2. Exam Topics

The final exam will last 2 and a half hours, and will consist of eight questions (after the traditional first question requiring your name and net-id). These eight questions are as follows:

**Class Implementation and Object Diagrams.** This question will be similar to questions 2 and 3 from the last prelim (now as a single question). You will be expected to finish the implementation of an incomplete class. You will then be given a sequence of assignment statements regarding this class; you are to diagram the memory representation (e.g. heap space, global space, etc.) for these statements.

**Call Frames.** You will be given the definition of one or more functions. You will be expected to draw the call frame (for a single function) or a call stack (for more than one function). The question might ask you to draw a call frame/stack at a single point in time (such as question 5b on the second prelim) or as it evolves over time (such as question 5a on the first prelim).

**Recursion.** You will be given the specification of a recursive function and asked to implement it. You should look at the review slides as well as question 5 from the previous prelim. We suspect that the recursive function is going to be a little harder (but not too much harder) than the second prelim.

**While-Loops.** You will be given a specification for a function that involves a while-loop (as well as some skeleton code). You are to specify the loop invariant and implement the body of the loop. The loop invariants for this question will be straight forward. Look at the loop invariants given in Lecture 21.

**Required Algorithm.** You will be given the precondition and post condition for one of the required algorithms. You are to draw the invariant using the horizontal list notation shown in class. Then you are to implement that algorithm.

**Multidimensional Lists.** We talked about multidimensional lists way back in Lecture 10. But you really did not get any exposure to them until Assignment 6. You will be given a specification on a multi-dimensional list and asked to implement it. This implementation will require two nested loops, just like the ones you wrote for Assignment 6.

**Exceptions and Try-Except.** This question will be similar to question 3b on the first prelim, but with the new except statements using dispatch-on-type. This was included in the review session for the second prelim, but was not put on the prelim itself. We promised it would be on the final, which it will be.

In addition, you should be prepared to write a small bit of code that uses a try-except block, just as you did in the function `fix_bricks` from Assignment 7.

**Short Answer.** We had debated on whether or not there would be a section of short answer questions on this exam. In the end, we decided to add them, though this section will be much shorter than on previous exams. We have included a complete study guide on terminology below.

## 3. Required Algorithms

On the final exam, you will be asked about one of the following algorithms:

- Binary Search
- Dutch National Flag
- Partition Algorithm
- Insertion Sort
- Selection Sort

You must know the specifications of these algorithms, we expect you to memorize them. Each of these algorithms involve a while-loop in a major way. Hence the specification consists of a precondition and postcondition.

You will be asked to develop a loop invariant from the specification. You are then expected to write the algorithm in such a way that you satisfy the invariant as well as the four "loop questions". That is,

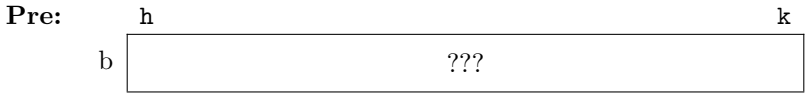
- How does loop start (how to make the invariant true)?
- How does it stop (is the postcondition true)?
- How does repetend make progress toward termination?
- How does repetend keep the invariant true?

We do *not* provide the implementations below; you are to come up with them on your own. In essence, part of this exam is how well you prepared this outside of class. A implementation that is given without an invariant, or which uses a different invariant will be considered wrong.

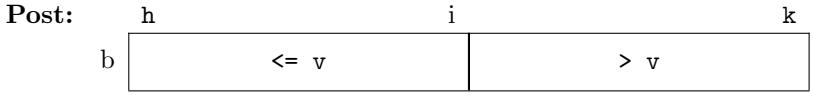
In all of the algorithms we may specify that the algorithm is to work with an array `b[0..b.length-1]`, or a segment `b[h..k]` or a segment `b[m..n-1]`. It should not matter which; you should prepare to work with whatever is given.

While we provide each of the invariants below, we highly recommend that you do not memorize the invariant. You should try to figure out the invariants on your own from the precondition and postcondition, as described in class.

**Binary Search.** The *vague* specification of binary search is to look for `v` in a sorted segment `b[h..k]`. A better specification is to assume that we start with the precondition



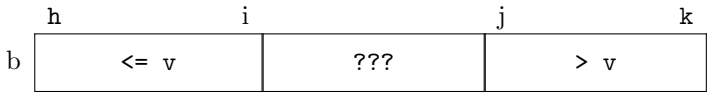

When binary search is done, you have a value `i` that satisfies the postcondition




Binary search returns `i` if `b[i] == v`. Otherwise, it returns -1.

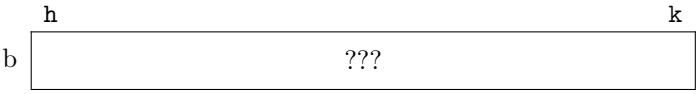
There is no one unique invariant for this problem. One acceptable invariant for this algorithm is as follows:

**Inv:**



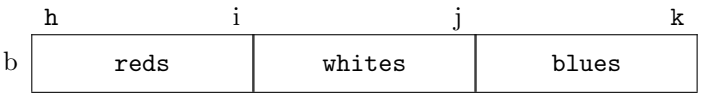

**Dutch National Flag.** For the Dutch National Flag, the array `b` contains objects which are colored *red*, *white*, *blue*. We start with the precondition

**Pre:**



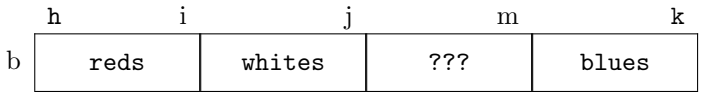

This means that we have no idea how the various colors are distributed about the array. The algorithm swaps the objects about to get objects of the same color close to one another. When done, we get the following postcondition, indicating that the array is "sorted":

**Post:**



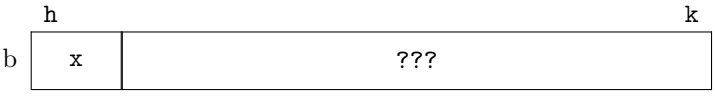

There is no one unique invariant for this problem. One acceptable invariant for this algorithm is as follows:

**Inv:**



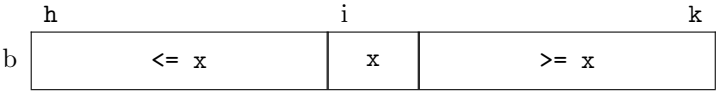

**Partition Algorithm.** In the partition algorithm, the array `b` comes with an initial value `x`:

**Pre:**



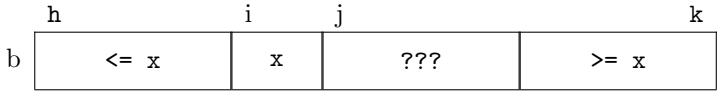

The algorithm swaps the elements of the array to achieve the following postcondition:

**Pre:**



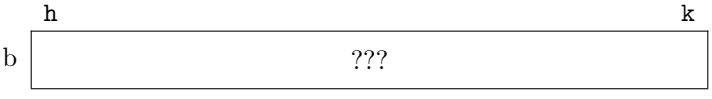

There is no one unique invariant for this problem. One acceptable invariant for this algorithm is as follows:

**Inv:**

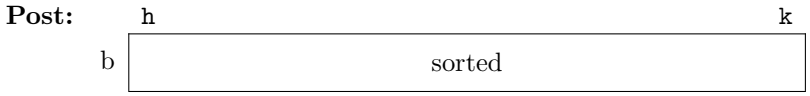



**Insertion Sort.** The *vague* specification is that insertion sort will sort the contents of an array (in ascending order). Indeed, when we give the precondition and postcondition, it is not much better. The precondition is

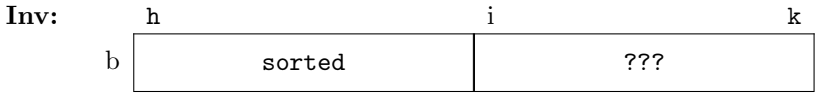
**Pre:**



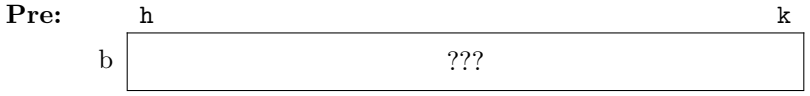

When insertion sort is done, we have an array that satisfies the postcondition



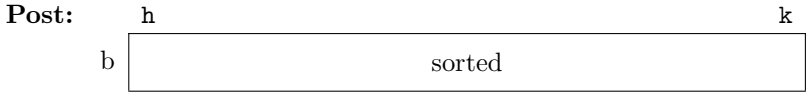

The important part of insertion sort (which distinquishes it from selection sort) is the invariant. The loop invariant for selection sort must have the following form:



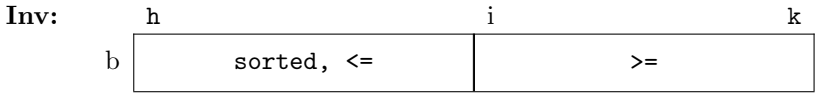

**Selection Sort.** Selection sort has the same specification as insertion sort. That is, it has precondition




and postcondition




The only difference is the invariant. The loop invariant for selection sort has an additional property, namely that everything in the left segment is less-than-or-equal to everything in the right segment.




## 4. Terminology and Important Concepts

Here, for your convenience is the list of terminology from the past two exams, as well as the new terminology since the last prelim. You should know the following terms, backward and forward. Wishy-washy definitions will not get much credit. Learn these not by reading but by practicing writing them down, or have a friend ask you these and repeat them out loud. You should be able to write programs that use the concepts defined below, and you should be able to draw objects of classes and frames for calls.

**Accumulator.** An accumulator is just a fancy name for a variable in a for-loop that stores information computed in the for-loop and which will be still available when the for-loop is complete.

*Example*: In the for looop `total = 0 for x in range(5):`

```
    total = total + x
```

the variable `total` is an accumulator. It stores the sum of the values 0..4.

**Active Namespace.** The active namespace the collection of all *global variables* and *functions* defined in the primary module. The primary module is the one which has been run as an application; that is, the one you execute typing

```
python <module file>
```

If you are running python in the interactive prompt, the primary module is an invisible, unnamed module.

Because of how Python stores function names, you should never name a global variable the same name as a function in the same namespace. The active namespace also includes the contents of any namespace imported via the `from` command. You should be sure never to import namespaces this way if they have names that conflict with the active namespace.

You do not need the module prefix to access any variables or functions in the active namespace.

**Assert Statement.** A *statement* of the form

```
assert <boolean-expression>
```

or

```
assert <boolean-expression>, <string-expression>
```

If the boolean expression is true, an assert statement does nothing. If it is false, it produces an error, stopping the entire program. In the second version of assert, it uses the string expression after the comma as its error message. Assert statements are used to enforce assertions such as preconditions, post conditions, or invariants.

*Example*:

```
assert 1 > 2, 'My Message'
```

This command crashes Python (because 1 is not greater than 2), and provides `'My Message'` as the error message.

**Assertion.** An assertion is a **property of a program** (as in the traditional notion of property, not a Python `@property`) that is either true or false. It represents a claim that must be true if the code is running correctly at that point. Assertions may be implemented as *assert statements*, but they do not have to be; more often than not, they are implemented as comments.

**Attribute.** Attributes are either *fields* or *properties*, and act as variables that are stored inside of an *object*. Attributes can often be modified, though not always the case, particularly when they are a *property*. Attributes typically have *invariants* which are rules specifying how the attribute may be modified.

*Example*: If the variable `color` stores an `RGB` object, then `color.red` is the red attribute in this object.

**Attribute Invariant.** See *invariant*.

**Bottom-Up Rule.** This is the rule by which Python determines which method definition to use when a method is called. It looks at the object, and starts from the bottom class partition (the partition with no *subclass* partition). It looks for a method definition whose name matches the one given. If it does not find one, it progressively moves up to each subclass partition until it finds one. It uses the definition for the first matching method that it finds.

**Call Frame.** A call frame is a formal representation of that Python uses when you execute a *function call*. It contains the name of the function as well as all parameters and local variables. It has also an instruction counter that tracks the next line in the function that is to be executed. A call frame is deleted (e.g. erased) as soon as the call completes.

**Call Stack.** The call stack is all of the *call frames* of the currently executing function calls (e.g. the main function call and all of its helper functions). These call frames are arranged in a stack, with the original function up top, and the most recent function call at the bottom. If the current function calls a helper function, you add a new frame to the bottom. When a helper function completes, you remove the call frame from the stack.

**Class.** A class is any *type* that is not built-in to Python (unlike `int`, `float`, `bool`, and `str` which are built-in). A value of this type is called an *object*.

**Class Definition.** This is a template or blueprint for the objects (or instances) of the class. A class defines the components of each object of the class. All objects of the class have the same components, meaning they have the same attributes and methods. The only difference between objects is the values of their attributes. Using the blueprint analogy, while many houses (objects) can be built from the same blueprint, they may differ in color of rooms, wallpaper, and so on.

In Python, class definitions have the following form:

```
class <classname>(<superclass>):
    <field assignments>
    <property definitions>
    <constructor definition>
    <method definitions>
```

In most cases, we use the built-in class `object` as the *super class*.

**Constructor.** A constructor is a *function* that creates a *object* for a `class`. It puts the object in heap space, and returns the name of the object (e.g. the folder name) so you can store it in a variable. A constructor has the same name as the *type* of the object you wish to create.

When called, the constructor does the following:

- It creates a new object (folder) of the class, setting all the field values to their defaults.
- It puts the folder into heap space
- It executes the method `__init__` defined in the body of the class. In doing so, it
  - Passes the folder name to that parameter `self`
  - Passes the other arguments in order
  - Executes the commands in the body of `__init__`
- When done with `__init__` it returns the object (folder) name as final value of expression.

There are no return statements in the body of `__init__`; Python handles this for you automatically.

*Example constructor call (within a statement)* : `color = RGB(255,0,255)`

*Example* ***__init__*** *definition*:

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

**Default Argument.** A default argument is a value that is given to a parameter if the user calling the function or method does not provide that parameter. A default argument is specified by wording the parameter as an assignment in the function header. Once you provide a default argument for a parameter, all parameters following it in the header must also have default argumetns.

*Example*:

```
def foo(x,y=2,z=3):
    ...
```

In this example, the function calls `foo(1)`, `foo(1,0)`, `foo(1,0,0)`, and `foo(1,z=0)` are all legal, while `foo()` is not. The parameter `x` does not have default arguments, while `y` and `z` do.

**Dispatch-on-Type.** Dispatch-on-type refers to a function (or other Python command) that can take multiple types of input, and whose behavior depends upon the type of these inputs. Operator overloading is a special kind of dispatch-on-type where the meaning of an operator, such as +, *, or /, is determined by the class of the object on the left. Dispatch-on-type is also used by `try-except` statements.

**Duck Typing.** Duck typing is the act of determining if an object is of the correct "type" by simply checking if it has attributes or methods of the right names. This much weaker than using the `type()` function, because two completely different classes (and hence different types) could share the same attributes and methods. The name was chosen because "If it looks like a duck and quacks like a duck, then it must be a duck."

**Encapsulation.** Encapsulation is the process of hiding parts of your data and *implementation* from users that do not need access to that parts of your code. This process makes it easier for you to make changes in your own code without adversely affecting others that depend upon your code. See the definitions of *interface* and *implementation*.

**Field.** A field is a type of object attribute. Fields are created by assigning a value to a variable inside of the class definition.

*Example fields* ***minutes****,* ***hours***:

```
class Time(object):
    minutes = 0
    hours = 0
```

It is impossible to enforce invariants on fields as any value can be stored in a field at any time. Therefore, we prefer to make fields hidden (by starting their name with an underscore), and replacing them with *properties*.

**Global Space.** Global space is area of memory that stores global variables and function names. It is divided into the *active namespace* plus any imported *namespaces*. Values in global space remain until you explicitly erase them or until you quit Python.

**Heap Space.** Heap space is the area of memory that stores *objects* (e.g. folders) as well as function definitions. Objects and function definitions in heap space remain until you explicitly erase them or until you quit Python. You cannot access heap space directly. You access objects/functions with variables in global space or in a call frame that contain the name of the value in heap space.

**Implementation.** The implementation of a collection of Python code (either a module or a *class*) are the details that are unimportant to other users of this module or class. It includes the bodies of all functions or methods, as well as all hidden attributes and functions or methods. These can be changed at any time, as long as they agree with the specifications and invariants present in the *interface*.

**Inheritance.** Inheritance is the process by which an object can have a method or attribute even if that method or attribute was not explicitly mentioned in the class definition. If the class is a subclass, then any method or attribute is *inherited* from the superclass.

**Interface.** The interface of a collection of Python code (either a module or a *class*) is the information that another user needs to know to use that module or class. It includes the list of all class names, the list of all unhidden attributes and their invariants, and the list of all unhidden functions/methods and their specifications. It does not include the body of any function or method, and any attributes that are hidden. The interface is the hardest part of your program to make changes to, because other people rely on it in order for their code to work correctly.

**Instance.** This is a synonym for an *object*. An object is an instance of a class.

**Invariant.** An invariant is an *assertion* that must **always be true**. The two types of invariants that we have talked about in class are *attribute invariants* and *loop invariants*.

A attribute invariant is a property of an attribute in an instance (object) in a class. The property must be true of the object after the constructor is finished; indeed, one of the purposes of a constructor is to ensure that the attribute invariant is true. The attribute invariant must also be true before and after each call to an instance method on the object.

A loop invariant is a property of variables (local, parameters, or attributes) used by a loop. A loop invariant must be true both before and after a single execution of the body of the loop.

**is.** The `is` operator works like == except that it compares folder names, not contents. The meaning of the operator `is` can never be changed. This is different from `==`, whose meaning is determined by the special operator method `__eq__`. If `==` is used on an object that does not have a definition for method `__eq__`, then `==` and `is` are the same.

**isinstance.** The function call `isinstance(ob,C)` returns `True` if object `ob` is an instance of class C. This is different than testing the type of an object, as it will return `True` even if the type of `ob` is a subclass of C.

**List.** A list is a mutable *sequence* that can hold values of any type. Lists are represented as a sequence of values in square braces (e.g. $[a_1, a_2, \ldots, a_n]$). A list can also hold other lists as well; this is how Python represents mutli-dimensional lists and matrices. For example, [[1,2],[3,4]] is a 2x2 list in Python. See Lecture 10 for more information on multi-dimensional lists.

**Loop Invariant.** See *invariant*.

**Method.** Methods are functions that are stored inside of an *object*. They are define just like a function is defined, except that they are (indented) inside-of a class defintion.

*Example method* `toSeconds()`:

```
class Time(object):
    # class with attributes minutes, hours def toSeconds(self):
        return 60*self.hours+self.minutes
```

Methods are called by placing the object variable and a dot before the function name. The object before the dot is passed to the method definition as the argument `self`. Hence all method definitions *must have at least one parameter*.

*Example*: If `t` is a time object, then we call the method defined above with the syntax `t.toSeconds()`. The object `t` is passed to `self`.

**Namespace.** A namespace is the collection of all *global variables* and *functions* defined inside of a module. Because of how the namespace stores function names, you should never name a global variable the same name as a function in the same namespace. You access a namespace by including it with the `import` command. In order to use any of the variables or functions in a namespace, you have to preface with the module name.

*Example*: `math.pi`, `math.cos(0)`

**Object.** An object is a value whose type is a *class*. Objects typically contain *attributes*, which are variables inside of the object which can potentially be modified. In addition, objects often have *methods*, which are functions that are stored inside of the object.

**Operator Overloading.** Operator overloading is the means by which Python evaluates the various operator symbols, such as +, *, /, and the like. The name refers to the fact that an operator can have many different "meanings" and the correct meaning depends on the type of the objects involved.

In this case, Python looks at the class or type of the object on the left. If it is a built-in type, it uses the built-in meaning for that type. Otherwise, it looks for the associated special method (beginning and ending with double underscores) in the class definition.

**Overriding a Method.** In a subclass, one can redefine a method that was defined in a superclass. This is called *overriding* the method. In general, the overriding method is called. To call an overridden method `method` of the superclass, use the notation

```
super(<classname>,self).method(...)
```

where `<classname>` is the name of the current class (e.g. the subclass).

**Precondition.** A precondition is an *assertion* that is placed before the start of a segment of code (e.g. before a loop, or before the start of a function). It must be true in order for the code that follows to work correctly.

**Post Condition.** A post condition is an *assertion* that is placed after the completion of a segment of code (e.g. after a loop, or at the return statement in a function call). It is guaranteed to be true if the code segment that procedes it is correct as specified.

**Property.** A property is a special type of attribute. It is a pair of a getter and a setter, together with a field. A property **must have an associated field** to work correctly, as the field is where it stores its value. The getter and setter are special methods that enforce the invariants for the property.

The getter and setter are each a method with the same name as the property. Python needs a *decorator* before the method to know it is the getter or setter. The decorator before a getter is `@property`. The decorator before a setter is `@<property-name>.setter`. The setter needs the property name its decorator because it is optional (the getter is not), and Python has to know which getter to associate it with.

*Example property `red` in `RGB`*:

```
class RGB(object):
    _red = 0

    @property # getter
    def red(self):
        return self._red


    @red.setter # setter
    def red(self,value):
        self._red = value
```

In this example, `_red` is the (hidden) field associated with this property.

**Scope.** The scope of a *variable* name is the set of places in which it can be referenced. Global variables may be referenced by any function that which is either defined in the same module as the global variable, or which imports that module; however, they cannot be reassigned in the body of a function. The scope of a parameter or local variable is the body of the function in which it is defined. The scope of an attribute is the same as the scope of the object that contains that attribute.

**Sequence.** A sequence is a type that represents a fix-length list of values. Examples of sequences are *lists*, *strings*, and *tuples*.

**Subclass.** A subclass D is a class that extends another class C. This means that an instance of D inherits (has) all the attributes and methods that an instance of C has, in addition to the ones declared in D. In Python, every user-defined class must extend some other class. If you do not explicitly wish to extend another class, you should extend the built-in class called `object` (not to be confused with an object, which is an instance of a class). The built-in class `object` provides all of the special methods that begin and end with double underscores.

**Tuple.** A tuple is identical to a *list* except that it is *immutable*. The contents cannot be removed, expanded, or otherwise altered. Tuples are represented as a sequence of values in parentheses (e.g. $(a_1, a_2, \ldots, a_n)$ ).

**Try-Except Statement.** This is a statement of the form

```
try:
    <statements>

except:
    <statements>
```

Python executes all of the statements underneath `try`. If there is no error, then Python does nothing and skips over all the statements underneath `except`. However, if Python crashes while inside the `try` portion, it recovers and jumps over to `except`, where it executes all the statements underneath there.

*Example*:

```
try:
    print 'A'
    x = 1/0 print 'B'

except:
    print 'C'
```

This code prints out `'A'`, but crashes when it divides 1/0. It skips over the remainder of the try (so it does **not** print out `'B'`). It jumps to the except and prints out `'C'`.

There is an alternate version of try-except that only recovers for certain types of errors. It has the form

```
try:
    <statements>

except <error-class>:
    <statements>
```

Python executes all of the statements underneath `try`. If there is no error, then Python does nothing and skips over all the statements underneath `except`. However, if Python crashes while inside the `try` portion, it checks to see if the error object generated has class `<error-class>`. If so, it jumps over to `except`, where it executes all the statements underneath there. Otherwise, the error propagates up the call stack where it might recover in another `except` statement or not at all.

*Example*:

```
try:
    print 'A'
    x = 1/0 print 'B'

except ZeroDivisionError:
    print 'C'
```

This code prints out `'A'`, but crashes when it divides 1/0. The execution skips over the remainder of the try (so it does **not** print out `'B'`). Since the error is indeed a `ZeroDivisionError`, it jumps to the except and prints out `'C'`.

Suppose, on the other hand, the try-except had been

```
try:
    print 'A'
    x = 1/0 print 'B'

except AssertionError:
    print 'C'
```

In this case, the code prints out `'A'`, but crashes when it divides 1/0 and does not recover.

**Type.** A type is a set of values and the operations on them. The basic types are types **`int`**, **`float`**, **`bool`**, and **`str`**. The type **`list`** is like **`str`**, except that its contents are mutable. For more advanced types, see the definition of *class*.

**Variable.** Depending on how you wish to think about it, a variable is a name with associated value **or** a named box that can contain a value. We change the contents of a variable via an assignment statement. A variable is created when it is assigned for the first time. We have seen four types of variables in this class: global variables, local variables, parameters, and attributes.

A *global variable* is a variable which is assigned inside of a module, but outside of the body or header of any function. The variable `FREEZING_C` that we saw in the module `temperature.py` is an example of a global variable. Global variables last as long as Python continues to run.

A *local variable* is a variable which is not a parameter, but which is first assigned in the body of a function. For example, in the function definition

```
def before_space(s):
    pos = s.find(' ')
    return s[:pos]
```

`pos` is a local variable. Local variables only exist in the context of a *call frame*.

A *parameter* is a variable in the parentheses of a function header. For example, in the function header

```
def after_space(s):
```

the parameter is the variable `s`. Parameters also only exist in the context of a *call frame*.

An *attribute* is a variable that is contained inside of a mutable object. In a point object, the attributes are `x`, `y`, and `z`. In the RGB objects from Assignment 2, the attributes are `red`, `green`, and `blue`.