PREPARING FOR PRELIM 2

CS 1110: FALL 2013

This handout explains what you have to know for the second prelim. There will be a review session with detailed examples to help you study. To prepare for the prelim, you can (1) practice writing functions and classes in Python, (2) review the assignments and labs, (3) review the lecture slides, (4) **read the text**, and (5) memorize terminology listed below.

The prelim will *not* cover while loops. It covers material up to and including material in lecture on November 7th. The test will focus primarily on recursion, iteration, and classes (e.g. Assignments 4 and 5, as well as all related labs).

1. Exam Information

The exam will be held **Thursday, November 14th from 7:30-9:00 pm**. For reasons of space, we are split across mutliple rooms.

- Students with last names A – G meet in Olin 155
- Students with last names H – K meet in Olin 165
- Students with last names L – R meet in Olin 255
- Students with last names S – Z meet in Upson B17

1.1. Review Session.

There will be a review session on **Sunday, November 10th** at 4pm in a Kimball B11. It will cover material in this handout and explain the basic structure of the exam. It will also go over several sample problems to help you prepare for the exam.

2. Content of the Exam

In studying for this exam, you should be wary of looking at the past exams too closely. We completely changed the way we drew folders this year (the old way made sense for Java, but is not the proper way for Python). We are also **not putting properties on the exam**. Standard getters and setters are completely fair game, however.

Once again, there will be five questions, each of roughly equal weight. These questions will be taken from some combination of the following six topics:

Recursion. You will be asked to write a recursive function. It will be roughly the complexity of the recursive functions in the lab (e.g. lab 7); we will not have a Turtle question. The recustion question on the final (not prelim 2) for Fall 2012 is an excellent example of good recursion exam question.

When writing your recursive function, you should pay attention to the important points:

(1) precise specification
(2) base case(s)
(3) recursive case(s)
(4) progress toward termination.

You should also be prepared to draw a (short) call stack for a recursive function. Think of problem problem 5 (a) on the past exam (and not problem 5 (b)). We have not decided to add such a question yet, but in the past, the really short recursive functions are generally all-or-nothing. This type of question would allow us to test your knowledge of recursion even if you did not do so well on the programming part.

Iteration. You will be given a problem that you will need to use a for-loop to solve. You should know how to use a for-loop on a sequence if you are given one, or how to use `range()` if you are not given one. You should know how to use an *accumulator* if needed to perform calculations using a for-looop.

As part of this question, we might ask you to write a function that processes a 2-dimensional list. This would likely require two nested for-loops. If you have completed Part I of A5, you have a lot of experience doing this.

There will be no for-loop involving dictionaries. If we include dictionaries, it will be part of a terminology/potpourri question (should we include one).

Classes. You should know how to create a class that includes attributes, getters, setters, an initializer, and methods. You should know the names of the three most important built-in methods (e.g. `__init__`,`__str__`, and `__eq__`), but you do not need to know the names of any of the others. We expect you to be able to construct getters and setters for attributes given a class specification. However, you do not need to use properties for your getters and setters.

You should also know how to create a subclass, and how inheritance and overriding work in Python. You should expect to be given a base class and be asked to subclass it to provide additional functionality.

Diagramming Objects. You will be given a series of assignments and constructor calls. You will be expected to identify (1) the number of objects that are created, (2) draw folder representations of each of each object, and (3) draw folder representations of each class.

This is something that we have been doing all through class in lecture, but which you (admittedly) have not gotten a lot of practice with. But since we assume that you have been following lecture, we believe this is fair game for a question. There will be a sample question like this in the review session. We also talk about this question a bit more below.

Exceptions. You should know how to raise an exception, and how the new-and-improved try-except statements work. We will give you the exception hierarchy (what is a subclass of what) to help you with this problem.

Short Answer. The short answer questions will focus on terminology, particularly regarding object-oriented methodology. For this part of the test, we recommend that you review the text as well as the lecture slides. In addition, we have a provided a list of important terminology below.

The short answer questions may also include short, poutporri-style questions that were not long enough to merit a separate category of their own. For example, we might ask a few questions about dictionaries (not covered in any other part of the exam) here.

3. Diagramming Folders

We have been representing objects as folders all semester long. In the past we have giveb them to you. Now it is time to do it on your own. Like call-frames, folders are one of these "metaphores" that everyone handles a little differently (and the way we draw them now is different from 2012). And because Python is a bit new, we are still ironing out the details of exactly what we do and do not care about in these metaphors. Here is what we want to see in our folders.

Object Folders. Objects should look like a manilla folder. The tab of this folder is an identifier giving the name (which is usually a number) of the object. This is the object name. You are free to make the identifier anything you wish.

Class Folders. Class folders are drawn like object folders, but the tabs are on the right. Inside the tab you write the class name. There is no identifier number for a class folder.

The class folder should contain any class attributes (attributes defined by an assignment statement outside of the body of a method). They should also contain all the methods defined in that class. When listing a method, it is enough to give the name of the method. You can list the parameters if you want (we do in class), but *this is optional.* You will not lose any points if you fail to do so.

If one class is a subclass of another, you should draw an arrow from the subclass folder to the parent class folder. You never need to draw a folder for `object`. Classes that extend `object` will have no arrow coming from them.

Example. Suppose you are given the following classes.

```
class BB(object):
    """Instance attributes:
        _b:  [int]"""

    def getB(self):
        return self._b

    def setB(self,value):
        assert type(value) == int
        self._b = value

    def __init__(self,b=5):
        self.setB(b)

    def __eq__(self,other):
        ...
```

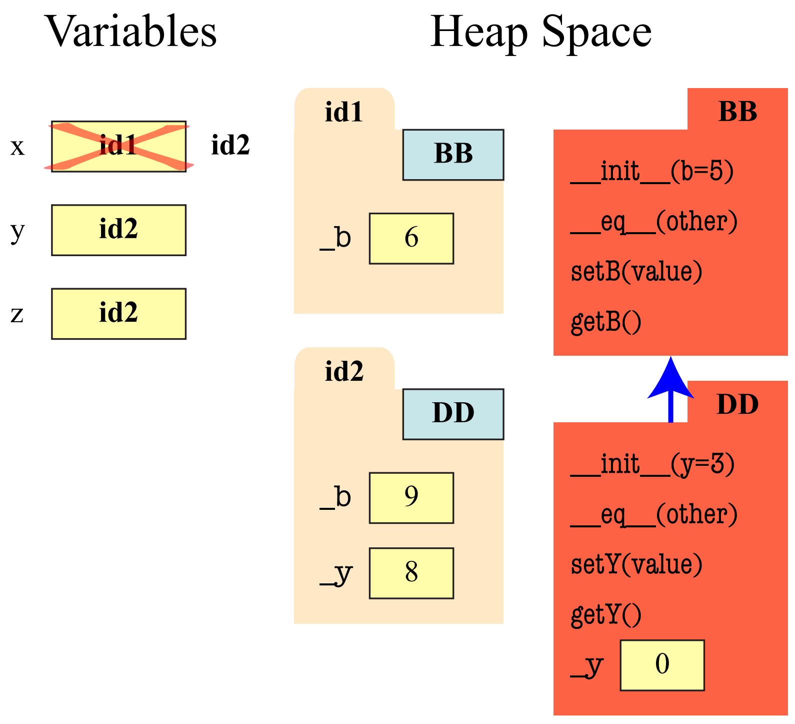
```
class DD(BB):
    _y = 0 # [int]

    def getY(self):
        return self._y

    def setY(self,value):
        assert type(value) == int
        self._y = value

    def __init__(self,y=3):
        BB.__init__(self,y+1)
        self.setY(y)

    def __eq__(self,other):
        ...
```

You are asked to "execute" the following sequence of commands:

```
>>> x = BB(6)
>>> z = DD(8)
>>> y = z
>>> x = y
```

There are three variables here and two constructor calls. You would draw two object folders – one for each constructor call. You would also draw two class folders – one for each class definition. All of the folders (both object and class) should be shown in heap space. The variables are drawn separately and belong in global space. You would put the names of each object folder in the variables. If a variable or attribute changes value, you should not create a new one. Instead, cross out the old value and replace it with a new one, just as we did with call frames.

With the example above, your answer might look something like this:



4. Terminology and Important Concepts

Below, we summarize the terms you should know for this exam. You should be able to define any term below clearly and precisely. If it is a Python statement, you should know its syntax and how to execute it. You should know all of this *in addition to* the terminology that you had to learn for the first prelim.

Accumulator. An accumulator is 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 loop

```
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.

Attribute. Attributes are variables that are stored inside of an *object*. Instance attributes belong to an object or *instance*. Instance attributes are created by assignment statement that prefaces the object name before the period. They are typically created in the class initializer.

Class attributes belong to the class. They are created by an assignment statement that prefaces the class name before the period. They are also created by any assignment statement in the class definition that is outside of a method definition.

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

Example: If the variable `color` stores an `RGB` object, then the assignment `color.red = 255` alters the red instance attribute. The assignment `RGB.x = 1` would create a class attribute `x`.

Bottom-Up Rule. This is the rule by which Python determines which attribute or method definition to use (when the attribute is used in an expression, or the method is called). It first looks in the object folder. If it cannot find it there, it moves to the class folder for this object. It then follows the arrows from child class to parent class until it finds it. If Python reaches the folder for `object` (the superest class of all) and still cannot find it, it raises an error.

If the attribute or method is in multiple folders, it uses the first one that it finds.

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>):
    <class specification>
    <getters and setters>
    <initializer 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 initializer 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 to determine whether to recover from an error.

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.*

Getter. A getter is a special method that returns the value of an instance attribute (of the same name) when called. It allows the user to access the attribute without giving the user permission to change it. It is an important part of *encapsulation.*

Example: If `_minutes` is an instance attribute in class `Time`, then the getter would be

```
class Time(object):
    def getMinutes(self):
        """Return:  _minutes attribute"""
        return self._minutes
```

Global Space. Global space is area of memory that stores any variable that is not defined in the body of a function. These variables include both function names and modules names, though it can include variables with more traditional values. Variables 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 *mutable objects* (e.g. folders). It also stores function definitions, the contents of modules imported with the `import` command, as well as class folders. Folders in heap space remain until you explicitly erase them or until you quit Python. You cannot access heap space directly. You access them with variables in global space or in a call frame that contain the name of the object in heap space.

Immutable Attribute. An immutable attribute is a hidden attribute that has a *getter*, but no *setter*. This implies that a user it not allowed to alter the value of this attribute. It is an important part of *encapsulation.*

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.

Invariant. An *invariant* is a statement about an attribute that must always be true. It can be like a precondition, in that prevents certain types of values from being assigned to the attribute. It can also be a relationship between multiple attributes, requiring that when one attribute is altered, the other attributes must be altered to match.

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.

Method. Methods are functions that are stored inside of an *class folder*. They are defined 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`.

Mutable Attribute. An mutable attribute is a hidden attribute that has both a *getter* and a *setter*. This implies that a user it allowed to alter the value of this attribute, provide that the invariant is not violated. It is an important part of *encapsulation*.

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).

Property. A property is a special way of creating *getters* and *setters*. A property **must have an associated attribute** to work correctly, as the atribute 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 defined 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):
        assert type(value) == int assert 0 <= value and value <= 255 self._red = value
```

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

Not all getters and setters are properties. Properties are an advanced topic.

Setter. A setter is a special method that can change the value of an instance attribute (of the same name) when called. The purpose of the setter is to enforce any invariants. The docstring of the setter typically mentions the invariants as a precondition.

Example: If `_minutes` is an instance attribute in class `Time`, then the setter would be

```
class Time(object):
    def setMinutes(self,value):
        """Set _minutes attribute to value

        Precondition:  value is int in range 0..59"""
        assert type(value) == int assert 0 <= value and value < 60 self._minutes = value
```

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.

Try-Except Statement (Limited). A limited try-except statement is a 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. It 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.