PREPARING FOR PRELIM 1

CS 1110: FALL 2012

This handout explains what you have to know for the first 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 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 recursion. It covers material up to and including material in lecture on September 25th. The test will require an understanding of both the basics of programming in Python.

1. Exam Information

The exam will be held **Thursday, October 4th from 7:30-9:00 pm**. For reasons of space, we are split across mutliple rooms.

- Students with last names A – P meet in Kennedy 1116
- Students with last names R – T meet in Warren 131
- Students with last names U – Z meet in Warren 231

1.1. Review Session.

There will be a review session on **Sunday, September 30**. The time and place will be announced later; watch this space. 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

This is a brand new course, and you do not have years of past exams to pull from. Therefore, to be fair, we are going to tell you exactly what is going to be on the exam. There will be five questions, each of roughly equal weight. These five questions will be as follows:

String Slicing. You will be given a specification for a function that takes a string as an argument, though it may have additional arugments. You will use your knowledge of string slicing to implement that function. This question will test skills that you developed in Assignment 1.

Call Frames. You will be given one more function definitions and a function call. You will be asked to draw the frame for the call. You may be asked to draw each executed step of the frame. You may be asked to draw a call stack of multiple frames. This question will test skills that you developed in Assignment 2.

Functions on Mutable Objects. You will be given a type for a mutable object (e.g. a class). The attributes of the mutable object will have invariants that limit what values can and cannot be assigned to them. You will then be given a function specification that you will need to implement; your implementation must respect the invariants. This question will test skills that you developed in Assignment 2.

Testing and Debugging. You will be given a function specification and asked to develop test cases for it. You may also be asked to implement traces to follow program flow. This question will test skills that you developed in Lab 3 and Assignments 1 and 3.

Short Answer. The short answer questions will focus on terminology and Python syntax. 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.

3. Terminology and Important Concepts

Below, we summarize the terms you should know. 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.

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.

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.

Assignment Statement. A *statement* of the form

```
<variable> = <expression>
```

If the variable on the left hand side does not exist yet, it creates the variable and stores the value of the expression inside. If the variable does exist, it replaces the old value of this variable with the value of the expression.

Attribute. Attributes are variables that are stored inside of an *object*. Attributes can often be modified, though not always the case. 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.

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). Like functions, classes are defined in modules, and we have to import the module to use values (e.g. *objects*) of that type.

Conditional Statement. A *statement* of the form

```
if <boolean-expression>:
    <statement>
    ...
    <statement>
```

or

```
if <boolean-expression>:
    <statement>
    ...
    <statement>
else:
    <statement>
    ...
    <statement>
```

The first form is executed as follows: if the boolean expression is true, execute the statements underneath; otherwise skip over them. The second form is executed as follows: if the boolean expression is true, execute the statements underneath the `if`; otherwise execute the statements underneath the `else`.

Example:

```
if 2 < 1:
    x = 3
else:
    x = 4
```

The variable `x` has value 4 when this conditional statement is executed.

There are additional forms of conditional statements using the keyword `elif` that were shown in class.

Constructor. A constructor is a *function* that creates a *mutable object.* 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. We have not yet seen how to define a constructor, but we know how to use one. A constructor has the same name as the *type* of the object you wish to create. Like fruitful functions, they are typically expressions and not statements.

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

Expression. An expression is Python code that produces a value. Expressions cannot be used by themselves; they must be put inside of a *statement.* Examples of expressions are values (e.g. `1`, `'Hello'`), complex expressions (e.g. `1+2`, ``'Hello '+`n` ``) and *fruitful functions* or *methods* (e.g. `round(n,0)`, `s.find('a')`)

Function. A function is a parameterized sequence of statements, whose execution performs some task. There are three kinds of functions: procedure, fruitful function, constructor. We also consider methods to be functions. See the definition of *method* for the difference.

A function should be followed by a docstring (`"""` ... `"""`) that says what the function does. This is called the specification. The specification has to be precise and clear. A potential user of the function should be able to look only at the comment and the list of parameters to know how to call the function; they should not have to look at the body of the function.

Fruitful Function. A fruitful function is one that performs some task and **returns a value**; because they return values they are typically expressions, and not statements. The statement **`return`** `<value>` is used to terminate execution of a function call and return `<value>`.

Example:

```
def max(x,y):
    """Returns:  the maximum of x and y
    Precondition:  x, y are floats
    if x >= y:
        return x
    return y
```

Example fruitful function call (within a statement) : `z = 1 + max(x,y);`

Function Call. : A function call is an invocation of the function with arguments. When a function is called, these arguments are placed into the parameter variables, and the body of the function is executed. A function call is associated with a *call frame* which stores the parameters and local variables as the body is being executed.

Global Space. Global space is area of memory that stores 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 *mutable objects* (e.g. folders). Objects 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.

Method. Methods are functions that are stored inside of an *object*. They can either be procedures or fruitful functions. They are called by placing the object variable and a dot before the function name.

Example: `find(s1)` is a method in all string objects. If the variable `s` is a string, then we call this method on `s` using the syntax `s.find(s1)`.

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. A (mutable) 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 (as opposed to be stored inside of a module).

Print Statement. A *statement* of the form

```
print <string-expression>
```

The expression evaluate to a value of type string. In this course, we use print statements for debugging and not much else.

Procedure. A procedure is a *function* that performs some task (and does not return a value). Procedures may be used as statements.

Example:

```
def greet(n):
    """Print a greeting to name
    Precondition:  name is a string"""
    print "Hello "+name+'!'
```

Example procedure call: `greet('Walker')`

Return Statement. A *statement* of the form

```
return <expression>
```

It is placed at the end of a fruitful function to return a value

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. The scope of a parameter or local variable is the body of the function in which it is defined. We do not worry about the scope of attributes for right now.

Statement. A statement is a command for Python to do something. We have seen the following five statements so far: assignment statements, return statements, assert statements, conditional-statements, and try-except statements. In addition, any *procedure* may be used as a statement.

Try-Except Statement. A *statement* of the form

```
try:
    <statement>
    ...
    <statement>
except:
    <statement>
    ...
    <statement>
```

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'`.

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. A variable is 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: parameters, local variables, global variables, and attributes.

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`.

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.

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.

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`.