PREPARING FOR THE FINAL EXAM

CS 1110: FALL 2019

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.

There is only one new problem on this exam: loop invariants. See the list of problems below for the types of questions that we might ask here. Note that while we expect you to be familiar with the various sequence algorithms, **we do not expect you to memorize any algorithms**.

1. Exam Information

The exam will be held **Tuesday, December 17th from 9:00-11:30 am** in Barton Hall. As many of you are aware, this is a huge room. It will be set up with tables, three to a table. We will occupy the *entire* room. Because we are all in the same area, we are not asking you to sort yourself by last name.

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.

The review session topics are as follows (time and rooms to be annouced later):

Wednesday, December 11$^{\mathbf{th}}$: Call Auditorium (Kennedy)

- Session 1: Call Frames and Diagramming Objects
- Session 2: Classes and Subclasses
- Session 3: Exceptions and Try-Except

Thursday, December 12$^{\mathbf{th}}$: Call Auditorium (Kennedy)

- Session 4: Lists and Sequences
- Session 5: Recursion
- Session 6: Open Question Session

Friday, December 13$^{\mathbf{th}}$: Call Auditorium (Kennedy)

- Session 7: Loop Invariants
- Session 8: Sequence Algorithms
- Session 9: Open Question Session

2. Exam Topics

The final exam will last 2 and a half hours, and will have seven questions (after the traditional first question requiring your name and net-id). This makes it two more questions than a normal prelim. These seven questions are chosen as follows:

Class Implementation and Object Diagrams. This question will be similar to Problems 4 and 5 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.

We will not require you to understand any of the advanced features like properties or class methods.

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 or as it evolves over time. Look at Problem 3 from the first prelim. You should be prepared to answer this question for a recursive function, such as the one in Problem 6 on the Fall 2018 final.

Recursion/Iteration. You will be given the specification of a function that requires iteration and/or recursion. You will be asked to implement that function. This question will be similar to Problems 2 and 3 from the second prelim. Please review all of the recursion questions on the previous exams.

Multidimensional-Lists. We have not had an exam question about tables. But you now have a lot of experience with them in Assignments 6 and 7. We can guarantee that there will be a question like this on the final. To study for this question, you should look at Problem 6 on the Fall 2014 final and Problem 4 on the Fall 2018 final. We will also have more examples at the lists review session on Wednesday.

Loop Invariants. You will be given a specification for a function that involves a while-loop (as well as some skeleton code). We will then ask you one (or possibly two) of three types of questions:

(1) Given a loop invariant, write the body of the loop so that it satisfies the invariant.
(2) Given a loop invariant and corresponding code, identify any errors that are in the code.
(3) Given a loop example with a postcondition and precondition, change it to match a new loop invariant.

For the first type of question, study Problem 6 of the Spring 2014 final. A question like this will be a simple function, closer to the functions that you worked on in Lab 11. We would never give a question like this for the functions in Lab 12.

For the second type of question, see Problem 6 on the Spring 2014 prelim 2. But the last type of question is the one I have talked about in class, and is generally my favorite. See the Problem 5 on the Fall 2016 final or Problem 7 on the Fall 2018 final for an example of this one.

Testing, Debugging, and Exceptions. You should review Problem 5 from the first prelim. You should also look at Problem 5 from Fall 2015 final. We have not had a long exam question on exceptions this year, so we are a bit overdue. They will be on this exam in some form or fashion. We may also ask you to write a small bit of code that uses a try-except block. See Problem 6 from the Fall 2017 final for an example of this.

Short Answer and Poutporri. This section will be similar to the first question from the first prelim. It will have short questions that do not belong anywhere else, focusing primarily on terminology.

3. 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 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.

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.

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 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`.

Attribute Invariant. See *invariant*.

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.

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>):
    <class specification>
    <getters and setters>
    <initializer definition>
    <method definitions>
```

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

Class Invariant. See *invariant*.

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, which is empty.
- 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 arguments.

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.

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 includes restricting access to attributes via getters and setters, but it also includes the usage of hidden methods as well. This process makes it easier for you to make changes in your own code without breaking the code of anyone who is using your class. 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):
        """Returns the 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.

The Heap. The heap or 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 the heap remain until you explicitly erase them or until you quit Python. You cannot access the heap 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. An implementation is a collection of Python code for a function, module, or class) that satisfies a specification. This code may be changed at any time as long as it continues to satisfy the specification.

In the case of a function, the implementation is limited to the function body. In the case of a class, the implementation includes the bodies of all methods as well as any hidden attributes or methods. The implementation for a module is similar to that of a class.

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 is the information that another user needs to know to use a Python feature, such as a function, module, or class. The simplest definition for this is any information displayed by the `help()` function.

For a function, the interface is typically the specification and the function header. For a class, the interface is typically the class specification as well as the list of all unhidden methods and their specifications. The interface for a module is similar to that of a class.

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

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.

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 13 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`.

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 `super` function as follows.

```
super().method(...)
```

where `method` is the name of the method being overridden.

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.

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.

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

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
```

Specification, Class. A class specification is description of the purpose of a class and how to use it. A class specification typically describes what entity the class is supposed to represent. Class specifications are often written using docstrings and include the class invariant.

Specification, Function (or Method). A function specification is a description of what a function should do. It should include (1) preconditions on the *arguments*, (2) the return value of the function (if it is a *fruitful function*, and any other details on what the function does. A function specification is typically written as a docstring comment.

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.

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`.