# PREPARING FOR PRELIM 2

CS 1110: FALL 2025

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 lectures and/or videos, and (4) memorize terminology listed below.

The prelim will covers material up to and including the lecture on November 5th (Abstraction). The test will focus on recursion, iteration, and classes (e.g. Assignments 4 - 6, as well as all related labs).

## 1. Exam Information

The exam will be held on **Tuesday, November 17th**, the Tuesday before Thanksgiving. Because of the size of the class, we are split across multiple classrooms. Rooms are assigned by the first letter of your last name (unless you are taking a make-up or are taking the exam through SDS). We are essentially using the same rooms as the previous exam.

- Students with last names A – O meet in Statler 185.
- Students with last names P – Z meet in Uris G01.

1.1. **Review Session.** There will be a review session on **Monday, December 1st** at a time to be announced at a later date. This review session 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

Once again, there will be (at most) 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. labs 15 and 16). We are unlikely to have a Turtle question. The recursion questions on Fall 2022 Prelim 2 are a good example of what we might ask: one straight-forward recusive function and one harder one (Recursion questions are often picked as the A-level question). Fall 2021 is another good one, though this time the harder question (`merge`) was the first one.

Your recursive function will either be a straight-forward recursive definition (think palindrome) or it will ask you to solve a divide-and-conquer problem. If it is the latter, remember the three steps:

(1) Solve the problem on small data
(2) Break up the problem and solve it (recursively) on the two halves
(3) Combine the answers back together to get the final answer

You should also be prepared to draw a (short) call stack for a recursive function. We have not decided to add such a question yet, but in the past, the really short recursive functions are generally all-or-nothing. A call stack 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()` function 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. Assignment 6 should have given you a lot of experience with this. We may also ask you to loop over dictionaries. Again at the 2022 Fall Prelim 2 for an example of this.

**Classes.** You should know how to create a class that includes attributes, getters, setters, an initializer, and very simple 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 invariant.

You should also know how to create a subclass, and should know 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. The labs are a good reference here.

**Diagramming Objects.** You will be given a series of assignments and constructor calls. You will be expected to (1) identify the number of objects that are created, (2) draw folder representations of each of each object, and (3) draw folder representations of each class. You should have a lot of experience with this after Assignment 5.

**Debugging.** There is enough material for this exam that we typically do not ask any short answer questions on this exam. However, there is a real possibility that (for the first time this year), we will ask a debugging question on this exam. If we ask such a question, we will do it at the cost of shortening either the iteration or recursion question. Such a debugging question would focus on code with for-loops. For an example of what such a debugging question might look like, see the 2025 Fall **Final Exam** for an example of this.

## 3. 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.

**Abstraction.** Abstraction is when one class pretends to be another through *operator overloading* or familliar looking methods. For example, you can use the methods `__len__`, `__getitem__`, and `__setitem__` methods to make your class act like a list. Abstraction helps a user work with a complicated concept (like image processing) by presenting it to the user in a simple-to-understand form.

**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 an empty object folder.
- 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.

**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.

**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):
    def toSeconds(self):
        """Returns minutes, hours as seconds"""
        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().method(...)
```

If you want to access the method in a class *other* than the immediate parent, use

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

where `<childclass>` is the *immediate child* of the class you want to access.

**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.