DALHOUSIE
UNIVERSITY

Master of Engineering in Internetworking

Lab # 6

Classes (Objects, Functions and Methods)

INWK 6312

Section A

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

attribute: One of the named values associated with an object.

embedded (object): An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

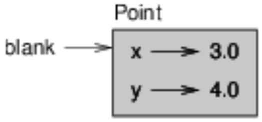
```
class Point(object): #creates a new class
    """Represents a point in 2-D space."""

>>> print Point
<class '__main__.Point'>

>>> blank = Point() # creating an instance of a “Point” object
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>


>>> blank.x = 3.0
>>> blank.y = 4.0
```

Object Diagram

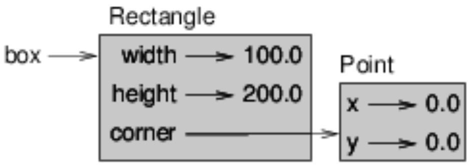


Question 1

Create a new class called Point. This class will have a “x” and “y” attribute.
write a function called `distance_between_points` *that takes two Points as arguments and returns the distance between them.*
Test you function by instantiating two instances and assigning them x and y attributes of type Int

Question 2

Create a new class called Rectangle, this class will have width, height and corner attributes. The corner attribute is an instance of the Point class created in Q1. The object diagram of an instance of the class is below:



- *Write a function called "find_center" that takes a Rectangle has an argument and returns a Point that returns a Point that contains the coordinates of the center of the Rectangle. (Assuming the corner of the rectangle is on the origin)*

- *Write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.*

- *Write a version of `move_rectangle` that creates and returns a new Rectangle instead of modifying the old one.*

Question 3

Swampy (see lab 2) provides a module named `World`, which defines a user-defined type also called `World`. You can import it like this:

```
from swampy.World import World
```

Or, depending on how you installed Swampy, like this:

```
from World import World
```

The following code creates a World object and calls the `mainloop` method, which waits for the user.

```
world = World()
world.mainloop()
```

A window should appear with a title bar and an empty square. We will use this window to draw Points, Rectangles and other shapes. Add the following lines before calling `mainloop` and run the program again.

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

You should see a green rectangle with a black outline. The first line creates a Canvas, which appears in the window as a white square. The Canvas object provides methods like `rectangle` for drawing various shapes.

`bbox` is a list of lists that represents the "bounding box" of the rectangle. The first pair of coordinates is the lower-left corner of the rectangle; the second pair is the upper-right corner.

You can draw a circle like this:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

The first parameter is the coordinate pair for the center of the circle; the second parameter is the radius.

If you add this line to the program, the result should resemble the national flag of Bangladesh (see `http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags`).

- *Write a function called `draw_rectangle` that takes a Canvas and a Rectangle as arguments and draws a representation of the Rectangle on the Canvas.*
- *Add an attribute named `color` to your Rectangle objects and modify `draw_rectangle` so that it uses the color attribute as the fill color.*
- *Write a function called `draw_point` that takes a Canvas and a Point as arguments and draws a representation of the Point on the Canvas.*
- *Define a new class called Circle with appropriate attributes and instantiate a few Circle objects. Write a function called `draw_circle` that draws circles on the canvas.*

SECTION B

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

planned development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

Question 1
Write a function called `print_time` *that takes a Time object and prints it in the form* `hour:minute:second`*. Hint: the format sequence* `'%.2d'`*prints an integer using at least two digits, including a leading zero if necessary.*

Question 2
Write a boolean function called `is_after` *that takes two Time objects,* `t1` *and* `t2`*, and returns* `True` *if* `t1` *follows* `t2` *chronologically and* `False` *otherwise. Challenge: don't use an* `if` *statement.*

Study and Understand the following code before continuing :

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum

def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Question 3

Write a correct version of `increment` that doesn't contain any loops. Write and use helper functions called time_to_int and int_to_time that can convert a Time object to seconds and seconds bac to to a Time object resepectively

Question 4

Write a function called `mul_time` *that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.*

Then use `mul_time` *to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).*

Question 5

The `datetime` *module provides* `date` *and* `time` *objects that are similar to the Date and Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at* http://docs.python.org/2/library/datetime.html.

1. *Use the* `datetime` *module to write a program that gets the current date and prints the day of the week.*
2. *Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.*

SECTION C

object-oriented language: A language that provides features, such as user-defined classes and method syntax, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

operator overloading: Changing the behavior of an operator like + so it works with a user-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

information hiding: The principle that the interface provided by an object should not depend on its implementation, in particular the representation of its attributes.

REFERENCE: BASIC OBJECT CUSTOMIZATIONS

This list is not exhaustive, visit the python documentation for more:
[https://docs.python.org/2/reference/datamodel.html - basic-customization](https://docs.python.org/2/reference/datamodel.html#basic-customization)

`object.`**`__new__`**(*cls*[, ...])

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of *cls*).

`object.`**`__init__`**(*self*[, ...])

Called after the instance has been created (by `__new__()`), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `BaseClass.__init__(self,[args...])`.

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customise it), no non-`None` value may be returned by `__init__()`; doing so will cause a **`TypeError`** to be raised at runtime.

`object.`**`__del__`**(*self*)

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

`object.`**`__repr__`**(*self*)

Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an "informal" string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

`object.`**`__str__`**(*self*)

Called by the `str()` built-in function and by the **`print`** statement to compute the "informal" string representation of an object. This differs from `__repr__()` in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

object.**__lt__**(*self*, *other*) , object.**__le__**(*self*, *other*), object.**__eq__**(*self*, *other*), object.**__ne__**(*self*, *other*), object.**__gt__**(*self*, *other*), object.**__ge__**(*self*, *other*)

These are the so-called “rich comparison” methods, and are called for comparison operators in preference to **__cmp__()** below. The correspondence between operator symbols and method names is as follows:

`x<y` calls `x.__lt__(y)`,

`x<=y` calls `x.__le__(y)`,

`x==y` calls `x.__eq__(y)`,

`x!=y` and `x<>y` call `x.__ne__(y)`,

`x>y` calls `x.__gt__(y)`, and

`x>=y` calls `x.__ge__(y)`.

object.**__cmp__**(*self*, *other*)

Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no **__cmp__()**, **__eq__()** or **__ne__()** operation is defined, class instances are compared by object identity (“address”). See also the description of **__hash__()** for some important notes on creating hashable objects which support custom comparison operations and are usable as dictionary keys

object.**__hash__**(*self*)

Called by built-in function **hash()** and for operations on members of hashed collections including **set**, **frozenset**, and **dict**. **__hash__()** should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

object.**__nonzero__**(*self*)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`, or their integer equivalents `0` or`1`. When this method is not defined, **__len__()** is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither **__len__()** nor **__nonzero__()**, all its instances are considered true.

object.**__unicode__**(*self*)

Called to implement **unicode()** built-in; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

Question 1

Rewrite `time_to_int` *(from Section B Q3) as a method. It is probably not appropriate to rewrite* `int_to_time` *as a method; what object you would invoke it on?*

Question 2

- *Write an init method for the* `Point` *class that takes* `x` *and* `y` *as optional parameters and assigns them to the corresponding attributes.*

- *Write a* `str` *method for the* `Point` *class. Create a Point object and print it.*

- *Write an* `add` *method for the Point class.*

Question 3

Write an `add` *method for Points that works with either a Point object or a tuple:*

- *If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.*
- *If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.*

Question 4

This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named `Kangaroo` *with the following methods:*

1. *An* __`init`__ *method that initializes an attribute named* `pouch_contents` *to an empty list.*
2. *A method named* `put_in_pouch` *that takes an object of any type and adds it to* `pouch_contents`*.*
3. *A* __`str`__ *method that returns a string representation of the Kangaroo object and the contents of the pouch.*

Test your code by creating two `Kangaroo` *objects, assigning them to variables named* `kanga` *and* `roo`*, and then adding* `roo` *to the contents of* `kanga`*'s pouch.*