INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney

Copyright Warning

Week 9: Classes and Objects

We will cover: What is a class, what is an object, encapsulation, `public` and `private`, creating and using Objects, reference type, methods, the `self` keyword

You should read: §§3.1 pg352-365

Lecture 17: Classes and Objects

Defining classes. Creating and using Objects

Types and Values

A type is a kind of a thing e.g. laptop is a type of computer, 3.14 is a type of a real number, tulip is a type of a flower.

Both `int` and `float` are *types* in Python.

With these types you can do many operations from anywhere in the program

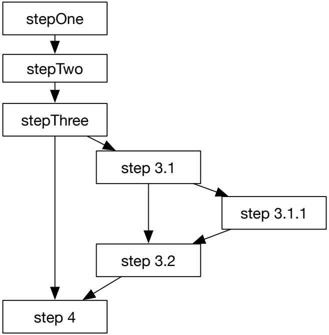
Reading or modifying a value of a type

```python
answer = 1 # must always be positive
data = get_input_data()
answer = 100 * data[0] # step 1

if data[1] % 2 == 0: # step 2
    data[2] += answer

while data[3] > 0: # step 3
    if data[3] <= 0:
        break
    if data[2] > 5: # step 3.1
        data[2] -= 1
    if answer <= 0:  # step 3.1.1
        answer += 1

    answer = answer + 1 # step 3.2
    data[3] -= 1

answer = calc_magic_flux( answer, 10000, 48.9 ) # step 4
print("Answer: " + str(answer))
```

Reading or modifying a value of a type

Organised into functions

```
answer = 0
data = get_input_data()
answer = step_one(data[0]) # do all steps
# calls -> answer = step_two(answer, data[1])
#    calls -> answer = step_three(answer, data[2], data[3])
#       calls -> answer = step_four( answer, 100 )
print("Answer: " + str(answer))
```



- When is `answer` required for read access?
- When is `answer` required for write access?
- ...
- If each task is done one of 4 persons, how do you control when answer is read or write?

Objects

We use variables in programming all the time, but there are cases when we need types which:

- control access to the value
- control the operations possible with the value
- contain more than one kind of value (not an array, but a composite of other types)

An object is a thing that may have data or particular methods, or (most commonly) *both*.

That means we have to *construct* them a bit differently.

Objects have Class

The *type* of an object variable is its *class*.

```
p = Point()
```

Objects are *instances* of things of a particular *Class*.

```
topleft = Point(-1 , -1)
right = Point(1, 0)
home = Point( -3388797, 15119390 )
```

Simple Classes

Let’s make a very simple class

```
class PositiveInteger:
    pass
```

We make an *instance* of this class like this:

```
x = PositiveInteger()
```

Simple Classes (cont.)

```python
class PositiveInteger:
    pass

x = PositiveInteger()
print(type(x))
```

```
<class ’__main__.PositiveInteger’>
```

I really want `x = PositiveInteger(53)`...this requires a special “method” called the *constructor*.

Constructors

The constructor does the work of building an instance of the class.

It creates space in memory for the object when it is called.

```python
class PositiveInteger:

    # constructor method for class
    def __init__(self):
        # things to do when creating
        # a new instance of class PositiveInteger
        self.number = 0
```

`self` is a keyword to describe the instance of the class (object). [1]

There is always a default constructor if none was defined

[1] `self` is also used to differentiate between function and method. Python: all methods are functions with first parameter being the associated object

Constructing a PositiveInteger object

We could have a very simple constructor for our new PositiveInteger class, like this:

```
class PositiveInteger:
    def __init__(self):
        # this constructor does nothing!
```

or this:

```
class PositiveInteger:
    def __init__(self):
        self.number = 1
        # this constructor changes the instance variable
```

or this:

```
    def __init__(self, initial_value):
        self.number = initial_value
        # this constructor changes the instance variable to the parameter
```

Constructors

A *constructor* for a *class* is a plan, or blueprint, to make an instance of the class.

When you make classes, new data types, you often need to make instances of them

`self` used to access the instance variable of the class.

What happens when you call the constructor?

Creating multiple instances.

```
1  class PositiveInteger:
2      def __init__(self, initial_value):
3          self.number = initial_value
4
5  water = PositiveInteger(80)
6  kelvin = PositiveInteger(5)
7  print("water is: " + str(water.number))
8  print("kelvin is: " + str(kelvin.number))
```

- What is happening at each step?
- How much memory is used?
- What would a diagram look like to describe the memory after executing line 7, but before line 8?

Creating an instance

When you do that,

- space is made for the object of type `PositiveInteger`
- the instance variable of the class is set to some value
- variables `water` and `kelvin` are ready for use.

What do you notice?

- no return type — it returns an object;
- PositiveInteger() is an expression. It will still create an object, even if it is not assigned.

```
PositiveInteger(35) # fine, but does nothing
print("some number is: " + str(PositiveInteger(35).number))
```

Using the instance

With an Object variable we use the dot . to refer to public instance variables or methods e.g.

```
text = "beetle"
uppertext = text.upper()
```

We can do the same with public instance variables, without parentheses
The instance variable value is visible from any part in the program with that

object reference.

Building larger programs

One program can have many classes

Here is a file `PositiveInteger.py`

```python
class PositiveInteger:
    def __init__(self, initial_value):
        self.number = initial_value
```

Here is a file `NegativeInteger.py`

```python
class NegativeInteger:
    def __init__(self, initial_value):
        self.number = initial_value
```

Here is a file `party.py`

```python
import PositiveInteger
import NegativeInteger

persons = PositiveInteger(10)
group_score = NegativeInteger(-10)
```

instance variables are public access by default

public instance variables are not helpful if we want to restrict read/write access

currently, any part of the program to modify the value without checking it is correct

```
import PositiveInteger
persons = new PositiveInteger(34)
persons.number = -470
print( "persons.number: " + str(persons.number) )
```

By design, any object of type `PositiveInteger` should contain only positive integer values

Object Methods

We can create many instances of the same class, but they each have their own memory

We can define a method of a class. A method is a function that associates with the memory of the object

```python
class PositiveInteger:
   def __init__(self, initial_value):
      self.number = initial_value

   def add_value(self, value):
      self.number += value

x = PositiveInteger(10)
x.add_value(4)
print(x.number)
```

Object Methods (cont.)

Beware

method vs function

```
    # method
    def add_value(self, value):

    # function
    def add_value(value):
```

instance variable vs local variable

```
    def add_value(self, value):
        # instance variable
        self.number += value

    def add_value(self, value):
        # local variable
        number += value
```

Read only methods

Methods associate with the memory of an object

```python
class PositiveInteger:
    def __init__(self, initial_value):
        self.number = initial_value

    def get_number(self):
        return self.number

    def calculate_offset(self, offset):
        '''returns a new number that is this number plus the offset. This does not modify the object memory'''
        return (self.number + offset)

x = PositiveInteger(10)
x.calculate_offset(4)
print(x.get_number())
```

The instance variable should not change. In fact, nothing about this object’s memory should change.

Read/Write methods

Constructor initialises that value when the object is created.

```python
class PositiveInteger:
    def __init__(self, initial_value):
        self.number = initial_value

    def get_number(self):
        return self.number

    def set_number(self, newnumber):
        self.number = newnumber

    def adjust_by_offset(self, offset):
        '''modifies the number to be this number plus the offset.'''
        self.number += offset

x = PositiveInteger(10)
x.adjust_by_offset(4)
print(x.get_number())
```

When does the value change over the lifetime of the object?

Controlling possible operations

Instead of access to the instance variable, we prefer methods

```python
import PositiveInteger

persons = PositiveInteger(5)
persons.set_number(-470)
print( "persons: " + persons.get_number() )
```

Isn’t this the same problem? can we prevent -470?

Controlling possible operations

The set method we define is one way the value can change from anywhere in the program

We can define restrictions on what possible values it can have

```
    def set_number(self, newnumber):
   # allowable number: zero or greater
      if newnumber >= 0:
           self.number = newnumber
```

Controlling possible operations (cont.)

Where else is it modified?

```python
class PositiveInteger:
    def __init__(self, initial_value):
        self.number = initial_value

    def get_number(self):
        return self.number

    def set_number(self, newnumber):
        self.number = newnumber

    def adjust_by_offset(self, offset):
        '''modifies the number to be this number plus the offset.'''
        self.number += offset
```

Controlling possible operations

We can further restrict how the values can change by having specific operations instead of `set_number()`

```
class PositiveInteger:
    number = 0
    def __init__(self, initial_value):
      # check initial value
        self.number = initial_value

    def get_number(self):
        return self.number

   def increment(self):
      self.number += 1

   def decrement(self):
      if self.number <= 0:
         return
      self.number -= 1
```

Controlling possible operations

Modified `party.py`

```
import PositiveInteger

persons = new PositiveInteger(2)
persons.decrement()
persons.increment()
persons.decrement()
persons.decrement()
persons.decrement()
persons.increment()
persons.decrement()
print( "persons: " + persons.get_number() )
```

Desk check the instance variable of `persons`