DePaul University
College of Computing and Digital Media

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, November 2, 2020

## Announcements

- Questions about the sixth assignment?
- The seventh assignment is due Thursday, November 5th – questions?

## Recursion

A **recursive method** is a method that solves a problem by making one or more calls to itself.

**Recursion** is a particularly helpful tool for certain problems.

Some programming languages are even primarily recursive in nature (e.g. Lisp and Scheme), where loops are almost completely absent from programs written in them.

Any recursive method consists of:

- **One or more base cases**: these are the portions of the problem for which an immediate solution is known
- **One or more recursive calls**: to avoid infinite recursion these subproblems must be in some way smaller than the original problem

The best way to learn recursion is to practice, a LOT. So we will see many examples during this part of the course.

## Fibonacci sequence

The **Fibonacci sequence** of integers is defined recursively as follows:

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

The sequence starts with 1, 1, 2, 3, 5, 8, 13, 21, 34, …

The obvious recursive algorithm can be found in **fib.py**.

The problem is that it can't compute even small(ish) Fibonacci numbers in a reasonable amount of time!

What happened?

There is a **huge duplication of effort** when the obvious recursive definition is implemented. It slows down the computation to point where large values become infeasible. Draw the picture!

Recursion is not always the right solution. See an iterative version that works efficiently in the file **fibonacci.py**.

## Traversal using recursion

One of the most natural uses of recursion is to traverse computer directories and web pages.

We'll start locally by considering how to recursively traverse computer directories.

### An anti-virus scanner

In the file **antiVirus.py** is the implementation of a simulation of an anti-virus scanner.

It stores the signatures of several (fake) viruses as a dictionary.

The name of the virus is the key and the signature of the virus is the value in each pair. (The names and signatures are fabricated).

The **high-level idea** of the program is as follows:

- Starting from the current directory (the folder in which antiVirus.py is stored), we **recursively visit all files and subdirectories** of the current directory.
- All the files discovered are **opened and checked** for the signature of the viruses in the dictionary. When a file with one of the signatures is found, a message is printed indicating which file was detected and where it was found.

The program uses several new Python features from the os module.

The **os module** contains functions that make available to the programmer operating systems features. To find what functions are available, type help(os) after importing the os module.

One of the functions the **antiVirus.py** program uses is os.listdir().

**os.listdir()** takes a string representing a folder pathname as a parameter and returns a list containing the names of entries in the folder. The names in the list are in arbitrary order.

Note: os.listdir() does not return a list of full pathnames, just a list of names of files in the folder described by the parameter.

Recursive calls on those files should be made on the files' pathnames, not the files' names.

The **pathname of the file/subfolder** is obtained by concentrating the folder with the name of the file or subfolder, for example on a Windows machine:

```
path = folder + "\" + file/subfolder
```

Explicitly doing the concatenation means that the program has to be changed if the type of machine on which it is being run changes.  Macs and Linux/Unix use "/" not "\".

A more portable solution is to use the **path.join()** method of the os module.

os.path.join(pathname, item) joins the two parameters to create a pathname that works on the machine on which the program is being executed.

Let's examine the **recursion in the anti-virus program** more carefully.

The **recursive call is on a subdirectory**.

If the item passed as a subdirectory is in fact a file, then the call to os.listdir() will throw an exception.

```
>>> os.listdir("test/directory.py")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    os.listdir("test/directory.py")
WindowsError: [Error 267] The directory name is
invalid: 'test/directory.py\\*.*'
```

If we put the call to os.listdir() into a try block, then we can put the base case in an associated except block.

In the **base case** we open the file, read its contents, and then check for all of the file signatures in our dictionary.

## Modifications to the anti-virus scanner

There are several modifications we can make on the anti-virus scanner:

1. Check for hidden files and avoid them (see **antiVirusForMacs.py**)

2. Explicitly check for file and directories and do the right thing (see **antiVirusRedux.py**).

## Extension of the approach

You can use these tools to solve a variety of problems having to do with files and directories.

**Exercise**: Write a recursive function **countFiles**() that counts and returns the number of files in a directory or any subdirectories of the directory.

You would use it as follows:

```
>>> countFiles('test')
10
>>> countFiles(os.path.join('test', 'test1'))
3
>>> countFiles(os.path.join('test', 'test2'))
4
>>> countFiles('test')
0
>>>
```

**Exercise**: Write a **recursive** function **search**() that takes as a parameter the name of a file and the pathname of a folder and searches for the file in the folder and any subfolder contained within it, directly or indirectly. The function should **return** the pathname of the file, if found, or None (the type in Python, not the string 'None') if the file cannot be found in anywhere in the folder or any of its subfolders. Capitalization of the file name should not matter when finding the file.

You would use it as follows:

```
>>> ans = search('dubious2.txt', 'test')
>>> ans
'test\\test1\\dubious2.txt'
>>> ans = search('dubious1.txt', 'test')
>>> ans
'test\\dubious1.txt'
>>> ans = search('amber.txt', 'test')
>>> ans
>>> type(ans)
<class 'NoneType'>
>>>
```

Find the solution in **dirProcessing.py**.

## Python objects

Anything that contains a value in Python is an object.

Each Python object has **three properties**:

1. An **identity**

2. A **type**
3. A **value**

The object's **identity** is the unique integer ID assigned to the object.

The ID is the object's memory address. The identity is accessible using the id() built-in function.

```
>>> a = 7
>>> id(a)
505493992
>>> lst = [3, 4, 5]
>>> id(lst)
31078800
>>> id(lst[0])
505493928
>>> id(lst[1])
505493944
>>> id(lst[2])
505493960
```

The **type** indicates what type the object is currently holding and what rules the objects are subject to.

The type of an object is accessible using the type() built-in function.

```
>>> type(lst)
<class 'list'>
>>> type(a)
<class 'int'>
>>> type('hello')
<class 'str'>
>>> type(None)
<class 'NoneType'>
```

The **value** is the data that the object represents.

```
>>> a
7
>>> lst
[3, 4, 5]
```

## Constructors

As we get ready for creating our own classes, it's useful to see that there is an alternative way to create new standard type instances.

The functions `int()` and `list()` are class (or type) constructors. When they are used with no parameters, they assign a default value to the variables they create.

```
>>> x = int()
>>> x
0
>>> id(x)
```

```
505493880
>>> type(x)
<class 'int'>
>>> lst = list()
>>> lst
[]
>>> type(lst)
<class 'list'>
```

If `int()` and `list()` are used with a parameter, they create a new variable with the value given as a parameter.

```
>>> y = int(3)
>>> y
3
>>> nums = list([1, 2, 3])
>>> nums
[1, 2, 3]
```

## Terminology

Every class is essentially a list of class **attributes**, meaning class variables and class functions (methods).

Each object of a type (i.e. each **instance** of a type) inherits the class attributes, that is, the class variables and functions.

The attributes of the type can be seen using the **help function**.

```
>>> help(int)
Help on class int in module builtins:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if
possible.  A floating
 |  point argument will be truncated towards zero
(this does not include a
 |  string representation of a floating point
number!)  When converting a
 |  string, use the optional base.  It is an error to
supply a base when
 |  converting a non-string.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __and__(...)
 |      x.__and__(y) <==> x&y
```

Etc.

A similar list can be obtained by writing: `help(list)`

## Operators

Again in preparation for writing our own classes, we need to reconsider operators.

There are some subtleties that we didn't consider before but that will be important when we are the people writing the class.

**Methods with names such as __x__** are special hooks that are called automatically when Python evaluates operators.

For instance if an object inherits an __add__ method, it is called when the object appears on the left side of a + expression.

See below for some examples:

```
>>> x = 3
>>> y = 4
>>> x.__add__(y)
7
>>> x + y
7
>>> x.__str__()
'3'
>>> str(x)
'3'
>>> lst = list([2, 3, 4, 5, 6])
>>> lst
[2, 3, 4, 5, 6]
>>> lst.__len__()
5
>>> len(lst)
5
>>> lst.__add__([4,5])
[2, 3, 4, 5, 6, 4, 5]
>>> lst
[2, 3, 4, 5, 6]
```

**Exercises**:

1. Define three integers and use the __x__ notation for the operators `abs`, `sub`, and `mul` on those integers.
2. Define three lists and use the __x__ notation for the operators `contains`, `len`, and `getitem`.

## Our first programmer-defined class

Classes provide a way for the application programmer to define new types.

Consider our first example, also found in the file **BankAccountStart.py**:

```
class Account(object):
    'a bank account class'
    def set(self, value):
        'set the balance to value'
        self.balance = value

    def get(self):
        'return the current balance on the account'
        return self.balance
```

There are a lot of things to explain here, but let’s first start by using this new type.

We can create a new object of type `Account` using the default constructor:

```
>>> acct = Account()
```

We can check that acct refers to an object of type `Account`, just to make sure:

```
>>> type(acct)
<class '__main__.Account'>
```

Once we have an `Account` object, we can invoke its class methods:

```
>>> acct.set(100)
>>> acct.get()
100
```

Finally, we can use the `help()` function just like we would on a built-in type:

```
>>> help(Account)
Help on class Account in module __main__:

class Account(builtins.object)
 |  a bank account class
 |
 |  Methods defined here:
 |  |
 |  get(self)
 |      return the current balance on the
account
 |
 |  set(self, value)
 |      set the balance to value
 |  |
 |  -------------------------------------------
---------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if
defined)
 |
 |  __weakref__
 |      list of weak references to the object
(if defined)
```

Note that because we wrote a docstring for every class function, the `help()` tool uses it as a part of the documentation.  We also defined a docstring for the class itself.

A user-defined type is **created using the `class` definition**:

```
class <Class Name> (<Super Class>):
```

Following the class keyword is `<Class Name>`, the name of the class. In parentheses and following the class name is some class `<Super Class>`.  This is the class that `<Class Name>` is inheriting attributes from.

In Python 3 and later it is recommended that a user-defined class **inherit from another already defined class** (whether built-in or user-defined).

If no existing class is suitable, then the built-in class `object` can be used.  The class `object` has no attributes at all and thus any class inheriting from it will inherit no class attributes.

```
>>> type(object)
<class 'type'>
>>> help(object)
Help on class object in module builtins:

class object
 |  The most base type
```

**A `class` statement** is not very different from a `def` statement.

A `class` statement defines a new type and gives the type a name.

A `def` statement defines a new function and gives the function a name.

The **assignments and function definitions inside a class statement** define the class attributes (i.e. the class variables and class methods).

**Class attributes** consist of class variables that determine the **state** of the class (type) and class functions that determine the class **behavior**, i.e. the functions that the class supports.

All **instances (objects) of the class are concrete objects** that inherit the class state and behavior.

We illustrate this by instantiating two objects we assign to variables acct1 and acct2:

```
>>> acct1 = Account()
>>> acct2 = Account()
```

Both objects inherit the two class methods:

```
>>> acct1.set(50)
>>> acct2.set(125.25)
>>> acct1.get()
50
>>> acct2.get()
125.25
>>>
```

The method `set` **takes as a parameter one argument** (`value`) and assigns it to the variable `balance`.

But if you look at the definition of the class, that doesn't seem right. The function `set` is defined to take two arguments (`self` and `value`), not just one!

We will explain the role of the first argument next.

Note for now that the variable `data` has a '`self.`' prefix, not unlike the prefix put in front of a module function.

The variable `balance` is NOT a class variable and it is NOT a local variable either (i.e. not local to the function `set`).

The variable `balance` is an instance variable whose scope and existence is tied to the instance. Every instance of the type `Account` will have its own balance variable.

In the above example, acct1 will have its own copy of `balance` and acct2 will have its own copy of `balance` as well.

As will be explained next time, every instance defines its own namespace. We can see this by accessing the data variable directly, by specifying the name space.

```
>>> acct1.balance
50
>>> acct2.balance
125.25
```

## Namespaces and classes

The general form of a Python class statement is as follows:

```
class <name> (superclass, …)  # assign to name
```

```
  data = value                       # shared class
variables
  def method (self, …)               # shared class methods
    self.member = value       # per-instance variables
```

It is very helpful to understand that in Python a class essentially defines a namespace.

The class attributes are just names defined in the namespace of the class.

For example, the class attributes of `Account` can be accessed by using `Account` as the namespace:

```
>>> Account.set
<function Account.set at 0x000002254C2817B8>
>>>
```

Further, **each instance object gets its own namespace and inherits class attributes** from the namespace of its class.

So when we instantiate a new instance of type `Account`, a new namespace gets created. It will inherit the attributes from the namespace of its class:

```
>>> acct = Account()
>>> acct.set
<bound method Account.set of <__main__.Account
object at 0x000002254C1FD860>>
>>>
```

When **Python works with classes, it really works with namespaces**.

Python automatically maps the invocation of a method by an instance of a class to a call to a function defined in the class namespace on the instance argument.

In other words, a method invocation like:

```
instance.method(arg1, arg2, …)
```

is translated by the Python interpreter into:

```
class.method(instance, arg1, arg2, …)
```

For example, instead of using

```
>>> acct.set(100)
```

you could use – and the Python interpreter does end up using after an automatic translation – the following:

```
>>> Account.set(acct, 100)
```

This explains **the self variable in the class definition**.

The self argument in the class methods refers to the instance making the method call, and the self.xxxx prefix in front of instance variables (or functions) refers to the instance namespace.

```
def set(self, value):
    self.balance = value
```

**Exercise 1**: Add a method to the class `Account` called `deposit`(). The method takes a parameter `value` and increases the balance by `value`.

```
>>> acct = Account()
>>> acct.set(100)
>>> acct.get()
100
>>> acct.deposit(50)
>>> acct.get()
150
>>> acct.deposit(50)
>>> acct.get()
200
>>>
```

**Exercise 2**: Add a method to the class `Account` called `withdraw`(). The method will take a parameter `value` and will reset the instance variable `balance` to remove `value` from the `balance`.

```
>>> acct = Account()
>>> acct.set(100)
>>> acct.get()
100
>>> acct.withdraw(50)
>>> acct.get()
50
>>> acct.withdraw(25.25)
>>> acct.get()
24.75
>>>
```

See **BankAccountSol.py** for the solutions.

## String representations for overloaded operators

There are two overloaded operators that return **the string representation of the object**: __repr__() and __str__().

The operator **__repr__()** is supposed to return the canonical string representation of the object.

This means that ideally, but not necessarily, if the function eval() is called with the string representation as input, it would return back the original object.

In other words, **eval(repr(o)) should give back the original object o**.

The `__repr__()` method is automatically called when an expression evaluates to an object in IDLE and this object needs to be shown in the IDLE window.

To illustrate the difference between the two, consider the following (artificial) class:

```
class Representation(object):
  def __repr__(self):
      return 'canonical string representation'
  def __str__(self):
      return 'pretty string representation'
```

It would be used as follows:

```
>>> rep = Representation()
>>> rep
canonical string representation
>>> print(rep)
pretty string representation
```

**Exercise**: Add `__repr__` and `__str__` methods to the `Account` definition. Make the display of the string method show a dollar sign and display two decimal places for the balance. See below for examples.

```
>>> acct = Account()
>>> acct.set(100)
>>> acct
Account(100)
>>> print(acct)
$100.00
>>>
```