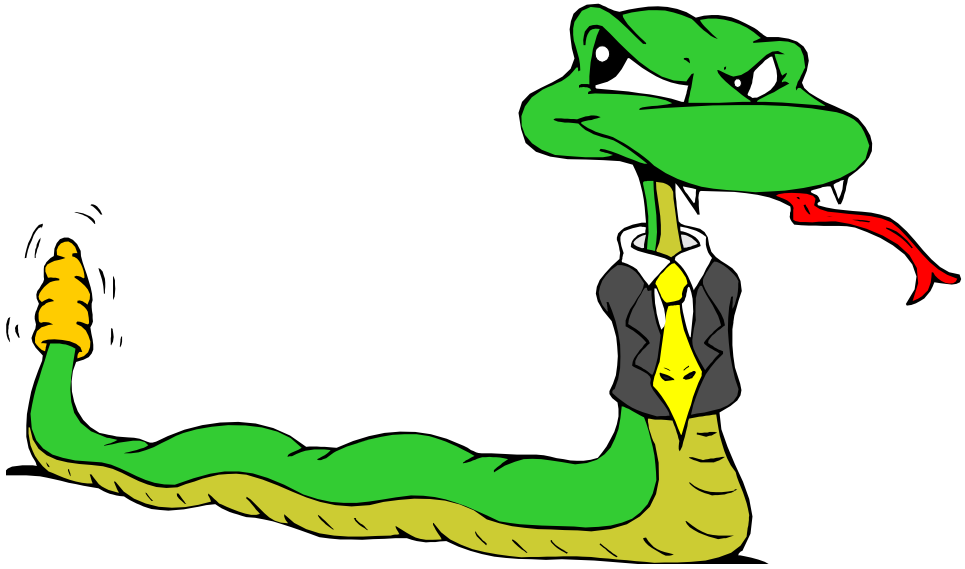
Functions in Python



Defining Functions

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword ‘return’ indicates the value to be sent back to the caller.

No header file or declaration of types of function or arguments

Calling a Function

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- Parameters in Python are *Call by Assignment*
  - Old values for the variables that are parameter names are hidden, and these variables are simply made to *refer to* the new values
  - All assignment in Python, including binding function parameters, uses *reference semantics.*

Function overloading? No.

- There is no function overloading in Python
  - Unlike C++, a Python function is specified by its name alone

    The number, order, names, or types of its arguments *cannot* be used to distinguish between two functions with the same name
  - Two different functions can't have the same name, even if they have different arguments
- But: see *operator overloading* in later slides

(Note: van Rossum playing with function overloading for the future)

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

Keyword Arguments

- You can call a function with some or all of its arguments out of order as long as you specify their names
- You can also just use keywords for a final subset of the arguments.

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Functions are first-class objects

Functions can be used as any other datatype, eg:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

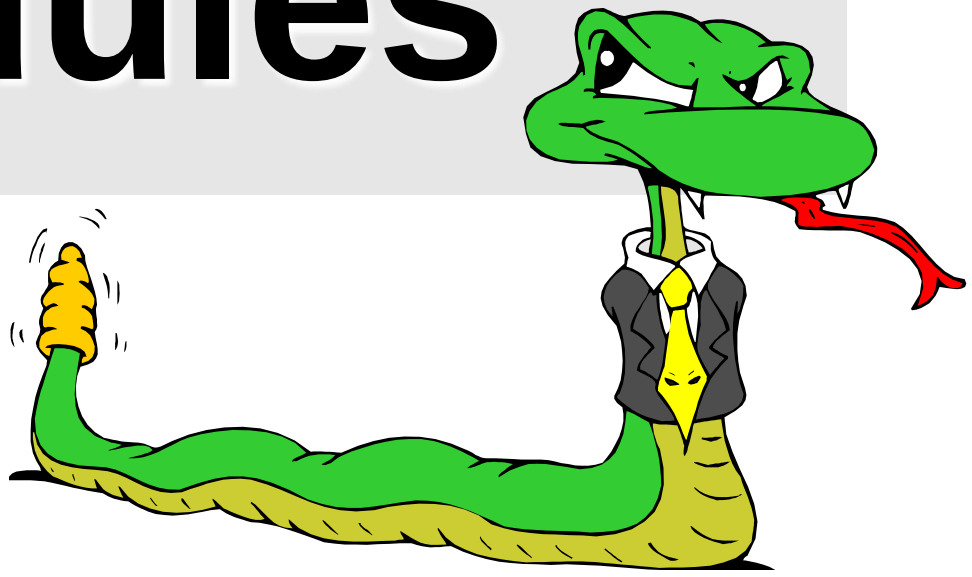
```
>>> def square(x):
        return x*x

>>> def applier(q, x):
        return q(x)

>>> applier(square, 7)
49
```

Lambda Notation

- Python uses a lambda notation to create anonymous functions

```
>>> applier(lambda z: z * 4, 7)
28
```

Importing and Modules



Importing and Modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

- The difference? What gets imported from the file and what name refers to it after importing

import …

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text “somefile.” to the front of its name:

```
somefile.className.method(“abc”)
somefile.myFunction(34)
```

```
import somefile as sf
```

from … import *

```
from somefile import *
```

- *Everything* in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- *Take care!* Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")
myFunction(34)
```

from … import …

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- After importing *className*, you can just use it without a module prefix. It's brought into the current namespace.
- *Take care*! Overwrites the definition of this name if already defined in the current namespace!

```
className.method("abc")
```
🡐 imported

```
myFunction(34)
```
🡐 Not imported

Directories for module files

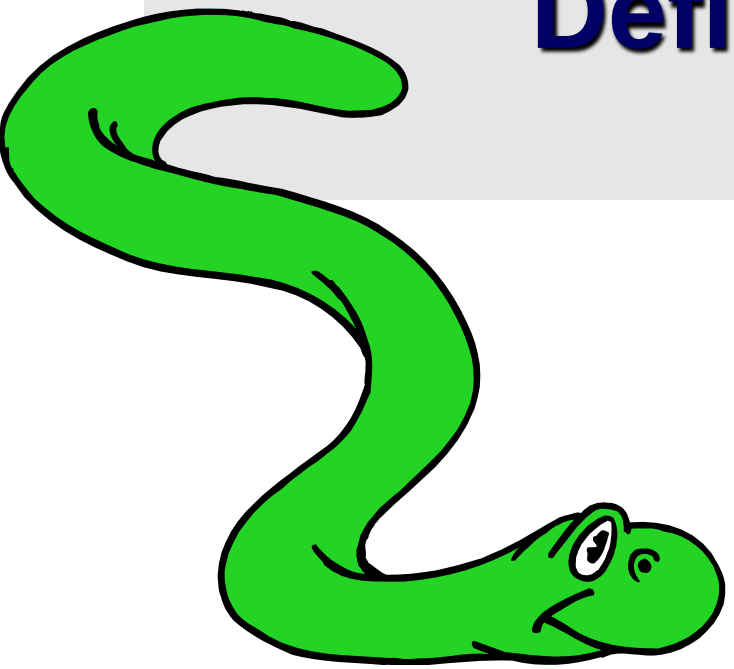
- *Where does Python look for module files?*
- The list of directories where Python will look for the files to be imported is sys.path
- This is just a variable named ‘path’ stored inside the ‘sys’ module

```
>>> import sys
>>> sys.path
['/Library/Frameworks/Python.framework/Versions/2.5/lib/python
    2.5/site-packages/setuptools-0.6c5-py2.5.egg', …]
```

- To add a directory of your own to this list, append it to this list

```
sys.path.append('/my/new/path')
```

Object Oriented Programming in Python: Defining Classes



Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- *Instances* are objects that are created which follow the definition given inside of the class
- Python doesn’t use separate class interface definitions as in some languages
- You just define the class and then use it

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block
- There must be a special first argument ***`self`*** in *all* of method definitions which gets bound to the calling instance
- There is usually a special method called ***`__init__`*** in most classes
- We'll talk about both later…

A simple class def: *student*

```
class student:
 """A class representing a
 student """
 def __init__(self,n,a):
      self.full_name = n
      self.age = a
 def get_age(self):
      return self.age
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student(“Bob”, 21)
```

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***`self`***
- In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*

Self

- Although you must specify *self* explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
student.set_age(x,23)
```

Access to Attributes and Methods

Definition of student

```
class student:
  """A class representing a student
  """
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student("Bob Smith", 23)

>>> f.full_name # Access attribute
"Bob Smith"

>>> f.get_age() # Access a method
23
```

Accessing unknown members

- Problem: Occasionally the name of an attribute or method of a class is only given at run time…
- Solution:

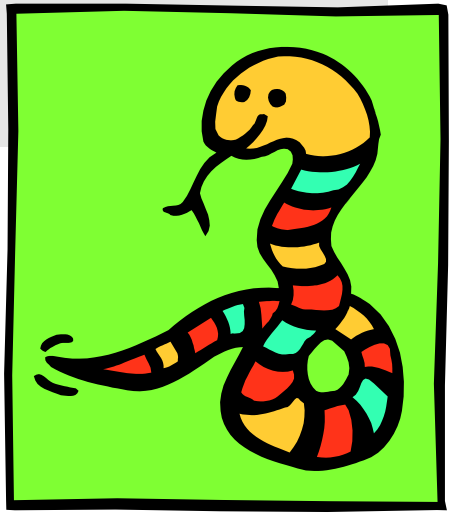
```
getattr(object_instance, string)
```

- **`string`** is a string which contains the name of an attribute or method of a class
- **`getattr(object_instance, string)`** returns a reference to that attribute or method

getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)
>>> getattr(f, "full_name")
"Bob Smith"
>>> getattr(f, "get_age")
 <method get_age of class
 studentClass at 010B3C2>
>>> getattr(f, "get_age")() # call it
23
>>> getattr(f, "get_birthday")
# Raises AttributeError - No method!
```

Attributes



Two Kinds of Attributes

- The non-method data stored by objects are called attributes
- *Data* attributes
  - Variable owned by a *particular instance* of a class
  - Each instance has its own value for it
  - These are the most common kind of attribute
- *Class* attributes
  - Owned by the *class as a whole*
  - *All class instances share the same value for it*
  - Called “static” variables in some languages
  - Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

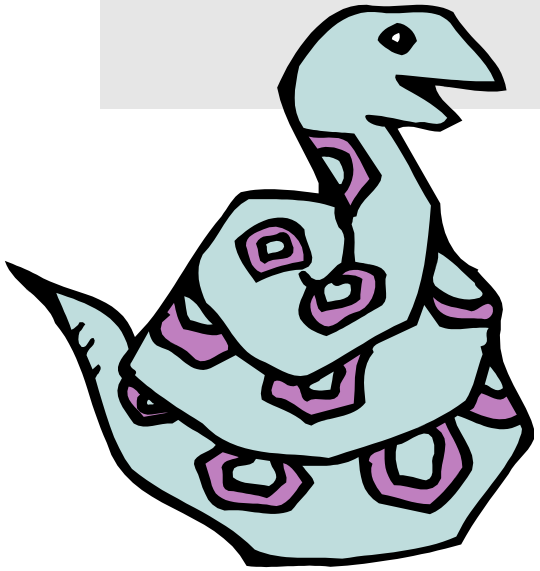
Class Attributes

- Because all instances of a class share one copy of a class attribute, when *any* instance changes it, the value is changed for *all* instances
- Class attributes are defined *within* a class definition and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
  - Access class attributes using `self.__class__.name` notation
    -- This is just one way to do this & the safest in general.
  - `Classname.name`

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Inheritance



Subclasses

- A class can *extend* the definition of another class
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
Class Cs_student(student):
```

  - Python has no 'extends' keyword like Java.
  - Multiple inheritance is supported.

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

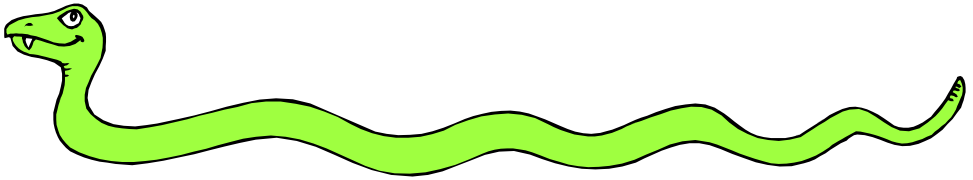
  - **The only time you have to explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Definition of a class extending student

```
Class Student:
  "A class representing a student."

  def __init__(self,n,a):
      self.full_name = n
      self.age = a

  def get_age(self):
      return self.age
```

```
Class Cs_student (student):
  "A class extending student."

  def __init__(self,n,a,s):
      student.__init__(self,n,a) #Call __init__ for student
      self.section_num = s

  def get_age():     #Redefines get_age method entirely
      print "Age: " + str(self.age)
```

Special Built-In Methods and Attributes



Built-In Members of Classes

- Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The built-in attributes define information that must be stored for all classes.
- All built-in members have double underscores around their names: `__init__` `__doc__`

Special Methods

- You can redefine these as well:

  `__init__` : The constructor for the class

  `__cmp__` : Define how **`==`** works for class

  `__len__` : Define how **`len(`** obj **`)`** works

  `__copy__` : Define how to copy a class

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call

Private Data and Methods

- Any attribute/method with 2 leading under-scores in its name (but none at the end) is **private** and can't be accessed outside of class
- Note: There is no 'protected' status in Python; so, subclasses would be unable to access these private data either.

Zadání úkolu

- Iniciativa TrueLibrary zahájila projekt na jediné a správné setřídění knih ve Velké knihovně. K tomu nutně potřebují znát vzájemnou podobnost knih.
- Vytvořte třídu TrueLib, která obsahuje seznam knížek (třídy Book) a atribut stopWords (a dále co bude potřeba☺)
- Třída Book obsahuje datové atributy název, autor, žánr (seznam/množina), rok a text knihy (shrnutí děje) a metody bagOfWords() a TF-IDF(), obojí s parametrem zda vynechávat stopwords
- Třída TrueLib dále obsahuje metodu generateSimilarity, která pro každé dvě knihy vypočte jejich podobnost jako (například) cosine similarity z bagOfWords, nebo TFIDF
  - Implementujte s pomocí list comprehension

1. Vypište 10 páru knih s nejvyšší podobností (název1, název2, podobnost)
2. Jak se výsledky mění při použití stopwords a BoW nebo TF-IDF
3. V rámci iniciativy TrueLibrary vznikla frakce starobehavioristů, kteří protestují proti použití obsahu knih při stanovování podobnosti. Vytvořte subclass TrueLib s předefinovanou podobnosti, která uvažuje pouze autora, žánr a rok vydání knihy (Jaccard + normalized distance?) (a vypište výsledky)

Zadání úkolu II.

- Stopwords například z http://xpo6.com/list-of-english-stop-words/
- BagOfWords = seznam slov s četností výskytu
  - TF ≈ BagOfWords
  - IDF = Inverse Document Frequency, logaritmus z poměru výskytu slova ve všech dokumentech, viz: https://cs.wikipedia.org/wiki/Tf-idf
- Jaccard Similarity $=(a \cap b)/(a \cup b)$
- Cosine Similarity = $$\frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$
- Setřídění seznamu: funkce sorted(), případně list.sort()
- Zdroj dat: booksummaries.txt
- Načítání dat: klíčové slovo with, funkce open()
  - Next(), read(), readline(),…
  - Metody stringu split(), replace(),…