Full Python Tutorial

- **Developed by Guido van Rossum in the early 1990s**
- **Named after Monty Python**
- **Available on eniac**
- **Available for download from http://www.python.org**

Python

- *Interpreted* **language: work with an evaluator for language expressions (like DrJava, but more flexible)**
- *Dynamically typed*: **variables do not have a predefined type**
- **Rich, built-in collection types:**
  - Lists
  - Tuples
  - Dictionaries (maps)
  - Sets
- **Concise**

Language features

- ***Indentation instead of braces***
- **Several sequence types**
  - Strings `'...'` : made of characters, immutable
  - Lists `[...]` : made of anything, mutable
  - *Tuples* `(...)` : *made of anything, immutable*
- ***Powerful subscripting (slicing)***
- ***Functions are independent entities (not all functions are methods)***
- **Exceptions as in Java**
- **Simple object system**
- **Iterators (like Java) and *generators***

Why Python?

- **Good example of *scripting language***
- **"Pythonic" style is very concise**
- **Powerful but unobtrusive object system**
  - *Every* value is an object
- **Powerful collection and iteration abstractions**
  - Dynamic typing makes generics easy

Dynamic typing – the key difference

- **Java: *statically typed***
  - Variables are declared to refer to objects of a given type
  - Methods use type signatures to enforce contracts
- **Python**
  - Variables come into existence when first assigned to
  - A variable can refer to an object of any type
  - All types are (almost) treated the same way
  - *Main drawback*: type errors are only caught at runtime

Recommended Reading

- **On-line Python tutorials**
  - The Python Tutorial (http://docs.python.org/tutorial/)
    - Dense but more complete overview of the most important parts of the language
    - See course home page for others
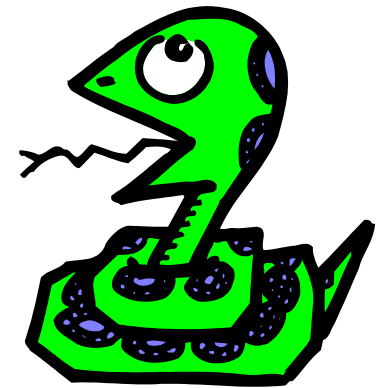
- **PEP 8- Style Guide for Python Code**
  - http://www.python.org/dev/peps/pep-0008/
  - The official style guide to Python, contains many helpful programming tips

- ***Many* other books and on-line materials**
  - If you have a specific question, try Google first

Technical Issues

Installing & Running Python

Which Python?

- **Python 2.7**
  - Current version on Eniac, so we'll use it
  - Last stable release before version 3
  - Implements some of the new features in version 3, but fully backwards compatible

- **Python 3**
  - Released a few years ago
  - Many changes (including incompatible changes)
  - *Much* cleaner language in many ways
  - Strings use Unicode, not ASCII
  - But: A few important third party libraries are not yet compatible with Python 3 right now

The Python Interpreter

- **Interactive interface to Python**

% python

Python 2.5 (r25:51908, May 25 2007, 16:14:04)

[GCC 4.1.2 20061115 (prerelease) (SUSE Linux)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

- **Python interpreter evaluates inputs:**

>>> 3*(7+2)
27

The IDLE GUI Environment (Windows)

12

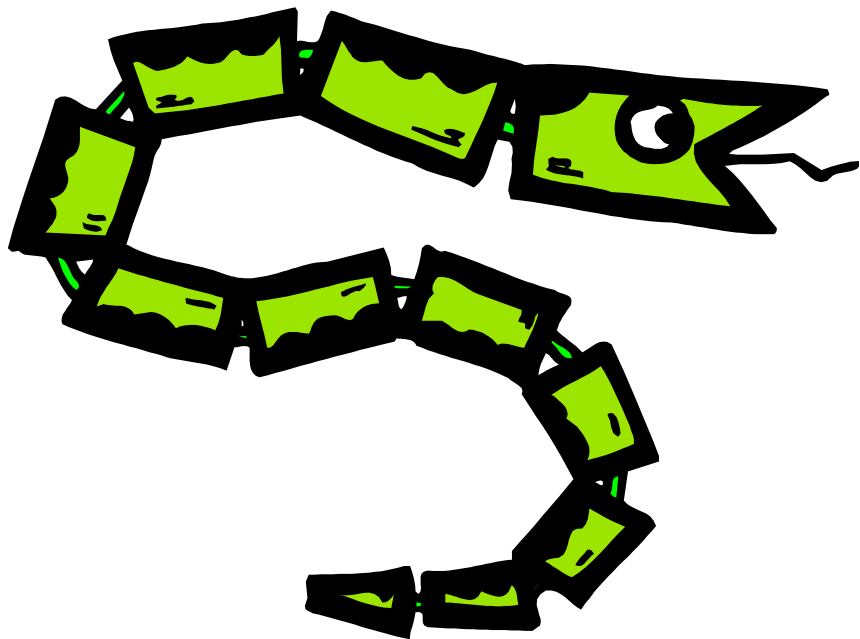# Running Programs on UNIX

% `python filename.py`

You can create python files using emacs.
(There's a special Python editing mode.
    `M-x python-mode`)

To make a python file executable, make this text the
    first line of the file :
#!/usr/bin/python

The Basics

A Code Sample (in IDLE)

```python
x = 34 - 23          # A comment.
y = "Hello"          # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"  # String concat.
print x
print y
```

Enough to Understand the Code

- **Indentation matters to the meaning of the code:**
    - Block structure indicated by indentation
- **The first assignment to a variable creates it.**
    - Variable types don't need to be declared.
    - Python figures out the variable types on its own.
- **Assignment uses $=$ and comparison uses $==$.**
- **For numbers $+ - * / \%$ are as expected.**
    - Special use of $+$ for string concatenation.
    - Special use of $\%$ for string formatting (as with printf in C)
- **Logical operators are words (`and, or, not`)** *not* **symbols**
- **Simple printing can be done with `print`.**

Basic Datatypes

- **Integers (default for numbers)**

  ```
  z = 5 / 2    # Answer is 2, integer division.
  ```

- **Floats**

  ```
  x = 3.456
  ```

- **Strings**

  - **Can use "" or '' to specify.**
    "abc" 'abc' (Same thing.)

  - **Unmatched can occur within the string.**
    "matt's"

  - **Use triple double-quotes for multi-line strings or strings than contain both ' and " inside of them:**
    """a'b"c"""

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- ***Use a newline to end a line of code.***
    - Use **** when must go to next line prematurely.
- **No braces { } to mark blocks of code in Python…** ***Use consistent indentation instead.***
    - *The first line with less indentation is outside of the block.*
    - *The first line with more indentation starts a nested block*
- **Often a colon appears at the start of a new block. (E.g. for function and class definitions.)**

Comments

- **Start comments with # – the rest of line is ignored.**

- **Can include a "documentation string" as the first line of any new function or class that you define.**

- **The development environment, debugger, and other tools use it: it's good style to include one.**

```python
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Assignment

- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*.

    - *Assignment creates references, not copies (like Java)*

- **A variable is created *the first time* it appears on the left side of an assignment expression:**

    ```
    x = 3
    ```

- **An object is deleted (by the garbage collector) once it becomes unreachable.**

- ***Names in Python do not have an intrinsic type. Objects have types.***

    - Python determines the type of the reference automatically based on what data is assigned to it.

(Multiple Assignment)

- **You can also assign to multiple names at the same time.**

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```
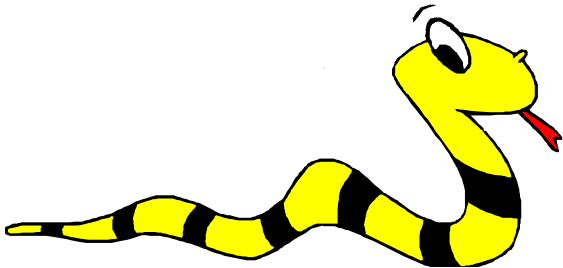
Naming Rules

- **Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.**

  ```
  bob   Bob   _bob   _2_bob_   bob_2   BoB
  ```

- **There are some reserved words:**

  ```
  and, assert, break, class, continue, def, del,
  elif, else, except, exec, finally, for, from,
  global, if, import, in, is, lambda, not, or, pass,
  print, raise, return, try, while
  ```

Sequence types:
Tuples, Lists, and Strings

Sequence Types

1. Tuple
   - A simple *immutable* ordered sequence of items
     - Immutable: a tuple cannot be modified once created....
   - Items can be of mixed types, including collection types

2. Strings
   - *Immutable*
   - Conceptually very much like a tuple
   - Regular strings use 8-bit characters. *Unicode strings* use 2-byte characters. (All this is changed in Python 3.)

3. List
   - *Mutable* ordered sequence of items of mixed types

Sequence Types 2

- **The three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.**

- **Tuples are defined using parentheses (and commas).**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- **Lists are defined using square brackets (and commas).**

```
>>> li = ["abc", 34, 4.34, 23]
```

- **Strings are defined using quotes (", ', or """).**

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types 3

- **We can access individual members of a tuple, list, or string using square bracket "array" notation.**
- ***Note that all are 0 based…***

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Optional argument allows selection of every n[th] item.

```
>>> t[1:-1:2]
('abc', (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

31

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1      # 2 names refer to 1 ref
                       # Changing one affects both

>>> list2 = list1[:]  # Two independent copies, two refs
```

The 'in' Operator

- **Boolean test whether a value is inside a collection (often called a *container* in Python:**

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- **For strings, tests for substrings**

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- **Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.**

The + Operator

- **The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.**
- **Extends concatenation from strings to other types**

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

Mutability:
Tuples vs. Lists

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
   ['abc', 45, 4.34, 23]
```

- **We can change lists *in place.***
- **Name *li* still points to the same memory reference when we're done.**

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

- *The immutability of tuples means they're faster than lists.*

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')   # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']


>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs the *+* operator.

- *+* creates a fresh list (with a new memory reference)
- *extend* is just like *add* in Java; it operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Confusing:
- *extend* takes a list as an argument *unlike Java*
- *append* takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')      # index of first occurrence*
1
```

   *more complex forms exist

```
>>> li.count('b')      # number of occurrences
2

>>> li.remove('b')     # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```
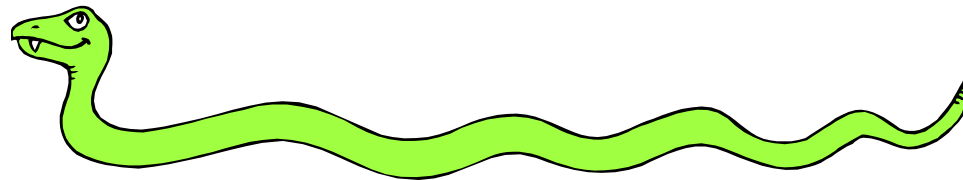
Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

Dictionaries: a *mapping* collection type

Dictionaries: Like *maps* in Java

- **Dictionaries store a** *mapping* **between a set of keys and a set of values.**
    - ***Keys can be any immutable type.***
    - Values can be any type
    - Values and keys can be of different types in a single dictionary
- **You can**
    - define
    - modify
    - view
    - lookup
    - delete

  the key-value pairs in the dictionary.

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- **Keys must be unique.**
- **Assigning to an existing key replaces its value.**

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- **Dictionaries are unordered**
    - **New entry might appear anywhere in the output.**
- **(Dictionaries work by *hashing*)**

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']            # Remove one. Note that del is
                             # a function.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                # Remove all.
>>> d
{}

>>> a=[1,2]
>>> del a[1]                 # (del also works on lists)
>>> a
[1]
```
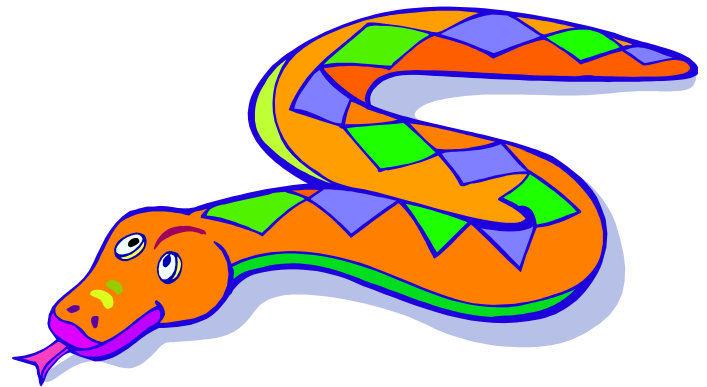
Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                    # List of current keys
['user', 'p', 'i']

>>> d.values()                  # List of current values.
['bozo', 1234, 34]

>>> d.items()        # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Boolean Expressions

True and False

- ***True* and *False* are constants**

- ***Other values are treated as equivalent to either True or False when used in conditionals:***
  - *False*: zero, *None*, empty containers
  - *True*: non-zero numbers, non-empty objects
  - See PEP 8 for the most Pythonic ways to compare

- **Comparison operators: ==, !=, <, <=, etc.**
  - `X == Y`
    - X and Y have same value (like Java *equals* method)
  - `X is Y`:
   - X and Y refer to the *exact same object* (like Java ==)

Logical Operators

- **You can also combine Boolean expressions.**
    - *True* if a is True and b is True: `a and b`
    - *True* if a is True or b is True: `a or b`
    - *True* if a is False: `not a`

Conditional Expressions

- `x = true_value` **`if`** `condition` **`else`** `false_value`

- lazy evaluation:
    - **First, `condition` is evaluated**
    - **If *True*, `true_value` is evaluated and returned**
    - **If *False*, `false_value` is evaluated and returned**

Control Flow
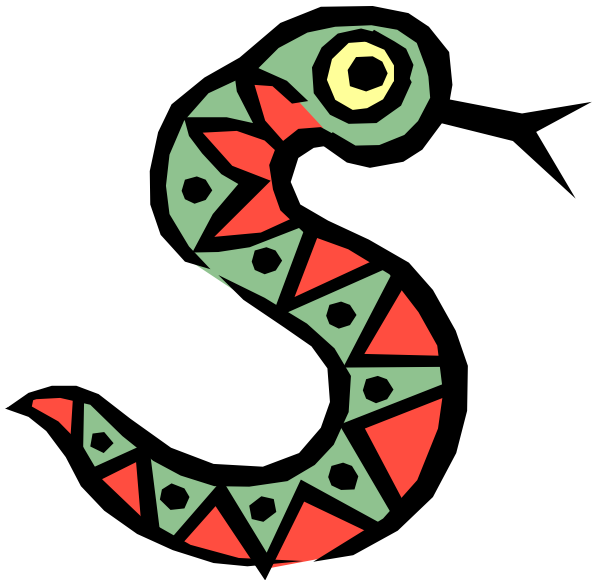
if Statements (as expected)

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

Note:
- **Use of indentation for blocks**
- **Colon (*:*) after boolean expression**

while Loops (as expected)

```
>>> x = 3
>>> while x < 5:
        print x, "still in the loop"
        x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
        print x, "still in the loop"

>>>
```

break and *continue*

- **You can use the keyword *break* inside a loop to leave the *while* loop entirely.**

- **You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and immediately go on to the next one.**
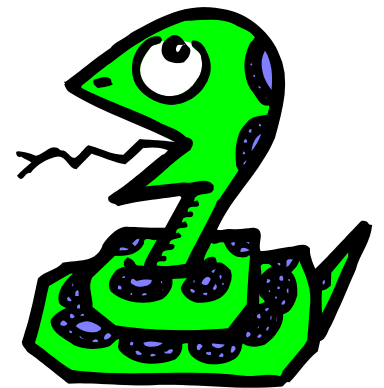
assert

- **An *assert* statement will check to make sure that something is true during the course of a program.**
  - If the condition if false, the program stops
    - (more accurately: throws an exception)

```
assert(number_of_players < 5)
```

- **Also found in Java; we just didn't mention it!**

For Loops

For Loops 1

- **For-each is Python's *only* form of for loop**
- **A for loop steps through each of the items in a collection type, or any other type of object which is "iterable"**

```
for <item> in <collection>:
    <statements>
```

- **If <collection> is a list or a tuple, then the loop steps through each element of the sequence.**

- **If <collection> is a string, then the loop steps through each character of the string.**

```
for someChar in "Hello World":
    print someChar
```

For Loops 2

```
for <item> in <collection>:
    <statements>
```

- **<item> can be more complex than a single variable name.**
  - If the elements of <collection> are themselves collections, then <item> can match the structure of the elements. (We saw something similar with list comprehensions and with ordinary assignments.)

```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
    print x
```

For loops and the *range()* function

- **We often want to write a loop where the variables ranges over some sequence of numbers. The *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.**

- **range(5) returns [0,1,2,3,4]**
- **So we can say:**

```
for x in range(5):
    print x
```

- **(There are several other forms of *range()* that provide variants of this functionality…)**
- ***xrange()* returns an *iterator* that provides the same functionality more efficiently**

Generating Lists using "List Comprehensions"
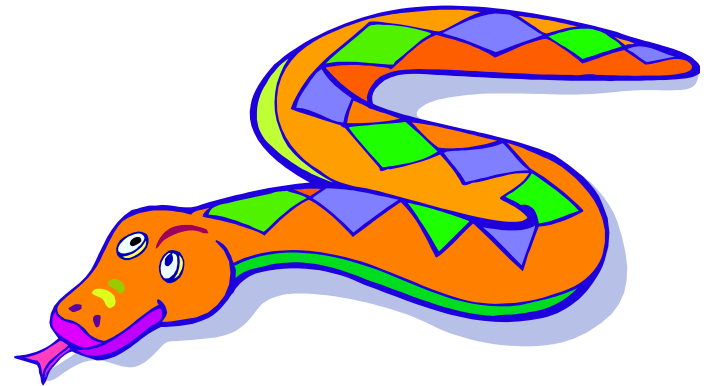
List Comprehensions 1

- **A powerful feature of the Python language.**
    - Generate a new list by applying a function to every member of an original list.
    - Python programmers use list comprehensions extensively. You'll see many of them in real code.

[expression for name in list]

List Comprehensions 2

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

[expression for name in list]

- **Where expression is some calculation or operation acting upon the variable name.**
- **For each member of the list, the list comprehension**
  1. **sets name equal to that member, and**
  2. **calculates a new value using expression,**
- **It then collects these new values into a list which is the return value of the list comprehension.**

65

List Comprehensions 3

- **If the elements of <u>list</u> are other collections, then <u>name</u> can be replaced by a *collection* of names that match the "shape" of the <u>list</u> members.**

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

Filtered List Comprehension 1

- **Filter determines whether expression is performed on each member of the list.**

- **When processing each element of list, first check if it satisfies the filter condition.**

- **If the filter condition returns *False*, that element is omitted from the list before the list comprehension is evaluated.**

Filtered List Comprehension 2

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- **Only 6, 7, and 9 satisfy the filter condition.**
- **So, only 12, 14, and 18 are produced.**

Nested List Comprehensions

- **Since list comprehensions take a list as input and produce a list as output, they are easily nested:**

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- **The inner comprehension produces: [4, 3, 5, 2].**
- **So, the outer one produces: [8, 6, 10, 4].**

For Loops / List Comprehensions

- **Python's list comprehensions provide a natural idiom that usually requires a for-loop in other programming languages.**
    - As a result, Python code uses many fewer for-loops

- *Caveat*! **The keywords *for* and *in* also appear in the syntax of list comprehensions, but this is a totally different construction.**

Functions in Python

(Methods later)
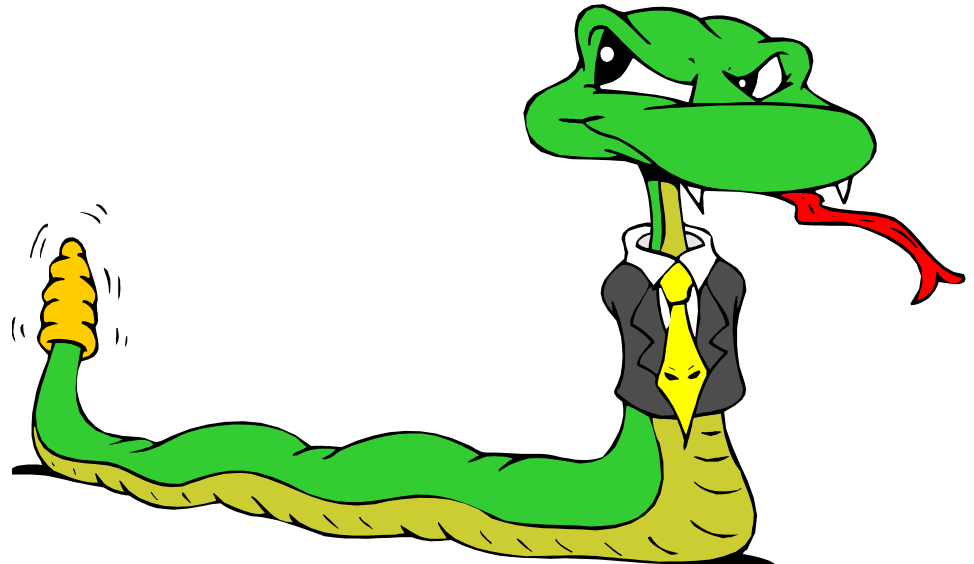
Defining Functions

Function definition begins with **def**

Function name and its arguments.

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

Colon.

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

'return' indicates the value to be sent back to the caller.

No declaration of <u>types</u> of arguments or result

Calling a Function

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

Functions without returns

- ***All* functions in Python have a return value**
  - even if no *return* line inside the code.

- **Functions without a *return* return the special value *None*.**
  - *None* is a special constant in the language.
  - *None* is used like *null* in Java.
  - *None* is also logically equivalent to False.
  - The interpreter doesn't print *None*

Function overloading? No.

- **There is no function overloading in Python.**
  - ***Unlike Java, 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 numbers of arguments.

  - *But **operator** overloading – overloading +, ==, -, etc. – is possible using special methods on various classes (see later slides)*

Functions are first-class objects in Python

- **Functions can be used just like any other data**
- **They can be**
    - **Arguments to function**
    - **Return values of functions**
    - **Assigned to variables**
    - **Parts of tuples, lists, etc**
    - **…**

```
>>> def myfun(x):
        return x*3

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

>>> apply(myfun, 7)
21
```

Lambda Notation

like anonymous
inner classes in Java

- **Functions can be defined without giving them names.**

- **This is most useful when passing a short function as an argument to another function.**

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

- **The first argument to apply() is an unnamed function that takes one input and returns the input multiplied by four.**

- **Note: only single-expression functions can be defined using this lambda notation.**

- **Lambda notation has a rich history in CS research and the design of many current programming languages.**

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)
```
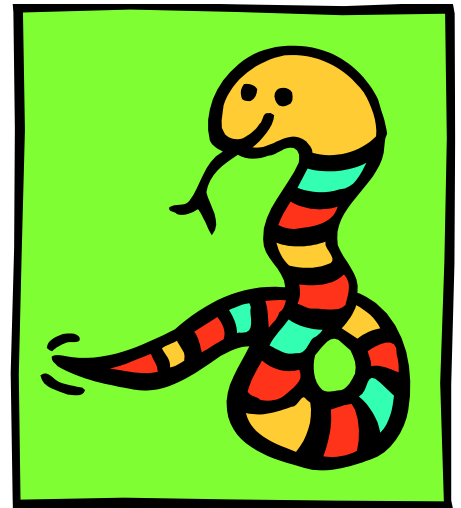
All of the above function calls return 8.

Keyword Arguments

- **Functions can be called with arguments out of order**
- **These arguments are specified in the call**
- **Keyword arguments can be used 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
```

Inheritance

Subclasses

- **A class can _extend_ the definition of another class**
    - Allows use (or extension) of methods and attributes already defined in the previous one.
    - New class: _subclass_. Original: _parent_, _ancestor_ or _superclass_

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

-

```python
class ai_student(student):
```

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

Redefining Methods

- **Very similar to over-riding methods in Java**

- **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 ever explicitly pass 'self' as an argument is when calling a method of an ancestor.

Extending __init__

- **Very similar to Java**
- **Commonly, the ancestor's `__init__` method is executed in addition to new commands.**
- ***Must be done explicitly***
- **You'll often see something like this in the `__init__` method of subclasses:**

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

Private Data and Methods

- **Any attribute or method with two leading underscores in its name (but none at the end) is private. It cannot be accessed outside of that class.**
    - Note:
    Names with two underscores at the beginning *and the end* are for built-in methods or attributes for the class
    - Note:
    There is no 'protected' status in Python; so, subclasses would be unable to access these private data either

Importing and Modules
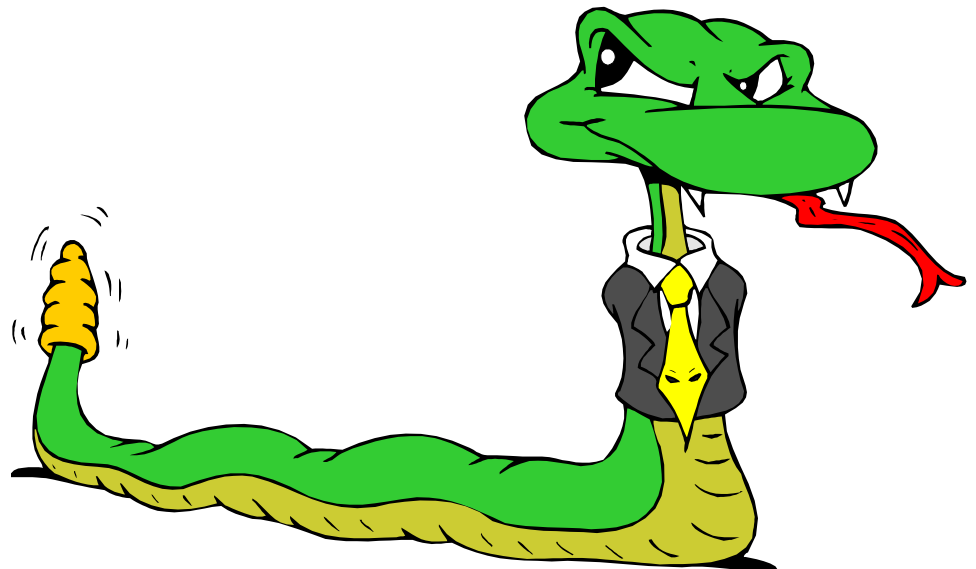
`Import` and Modules

- **Programs will often use classes & functions defined in another file**

- **A Python module is a single file with the same name (plus the *.py* extension)**

- ***Modules can contain many classes and functions***

- **Access using *import* (like Java)**

Where does Python look for module files?

- **The list of directories where Python looks: sys.path**

- **When Python starts up, this variable is initialized from the PYTHONPATH environment variable**

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

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

  - **Oops! Operating system dependent….**

Import I

`import` `somefile`

- ***Everything* in somefile.py can be referred to by:**

`somefile.className.method(`"`abc`"`)`

`somefile.myFunction(34)`

- `from` `somefile` `import` `*`

  - ***Everything* in somefile.py can be referred to by:**

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

  `myFunction(34)`

  - Careful! This can overwrite the definition of an existing function or variable!

Import II

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- Refer to it without a module prefix.
- *Caveat*! This can overwrite an existing definition.

```
className.method("abc")
```
← This was imported

~~myFunction(34)~~
← Not this one

Commonly Used Modules

- **Some useful modules, included with Python:**

- **Module: sys   - Lots of handy stuff.**
- **Module: os   - OS specific code.**
- **Module: os.path   - Directory processing.**

- **The Python standard library has *lots* of other useful stuff...**

More Commonly Used Modules

- **Module: math   - Mathematical functions.**
    - Exponents
    - sqrt

- **Module: Random   - Random numbers**
    - Randrange   (good for simulations, games, …)
    - Uniform
    - Choice
    - Shuffle

- **To see what's in the standard library of modules, check out the Python Library Reference:**
    - http://docs.python.org/lib/lib.html
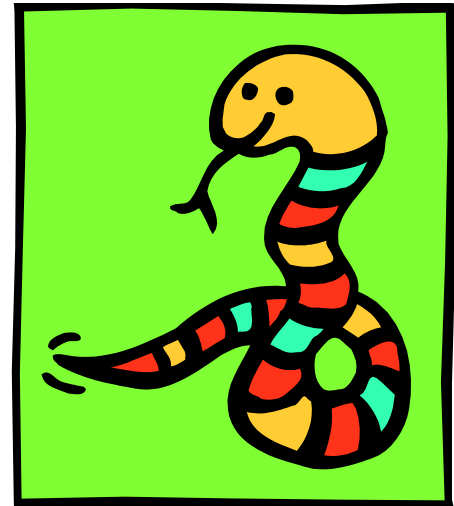
String Operations

String Operations

- **The string class provides a number of methods for useful formatting operations:**

  ```
  >>> "hello".upper()

   'HELLO'
  ```

- **Check the Python documentation for many other handy string operations.**

- **Helpful hint: use `<string>.strip()` to strip off final newlines from lines read from files**

String Formatting Operator: %

- **The operator *%* allows strings to be built out of many data items in a "fill in the blanks" fashion.**
  - Allows control of how the final string output will appear.
  - For example, we could force a number to display with a specific number of digits after the decimal point.

- **Very similar to the sprintf command of C.**

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- **The tuple following the *%* operator is used to fill in the blanks in the original string marked with *%s* or *%d*.**
  - Check Python documentation for details.

Printing with Python

- **print a string to the standard output stream using "print"**
- **Using the % string operator in combination with the print command, we can format our output text.**

```
>>> print  "%s xyz %d"  %  ("abc", 34)
abc xyz 34
```

"Print" automatically adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them.

```
>>> print "abc"
abc
```

```
>>> print "abc", "def"
abc def
```

- **Useful trick:**   `>>> print "abc",`   **doesn't add newline (does add space)**

String to List to String

- **Join turns a list of strings into one string.**

  <separator_string>.join(<some_list>)

  ```
  >>> ";".join( ["abc", "def", "ghi"] )
  "abc;def;ghi"
  ```

- **Split turns one string into a list of strings.**

  <some_string>.split(<separator_string>)

  ```
  >>> "abc;def;ghi".split( ";" )
  ["abc", "def", "ghi"]
  ```
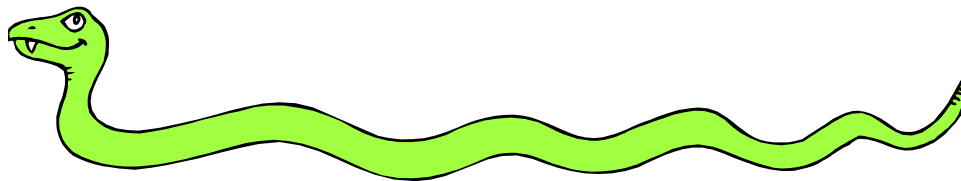
Convert Anything to a String

- **The built-in str() function can convert an instance of <u>any</u> data type into a string.**
    - You can define how this function behaves for user-created data types. You can also redefine the behavior of this function for many types.

```
>>> "Hello " + str(2)
"Hello 2"
```

96

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: `__doc__`**

Special Methods

- **For example, the method `__repr__` exists for all classes, and you can always redefine it.**
- **The definition of this method specifies how to turn an instance of the class into a string.**
  - `print f` sometimes calls `f.__repr__()` to produce a string for object f.

  - If you type `f` at the prompt and hit ENTER, then you are also calling `__repr__` to determine what to display to the user as output.

Special Methods – Example

```
class student:
    ...
     def __repr__(self):
        return "I'm named " + self.full_name
    ...


>>> f = student("Bob Smith", 23)
>>> print f
I'm named Bob Smith
>>> f
"I'm named Bob Smith"
```

Special Methods

- **Used to implement operator overloading**
  - Most operators trigger a special method, dependent on class

  `__init__` : The constructor for the class.
  `__len__` : Define how `len(`obj`)` works.
  `__copy__` : Define how to copy a class.
  `__cmp__` : Define how == works for class.
  `__add__` : Define how + works for class
  `__neg__` : Define how unary negation works for 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.**

Special Data Attributes

- **These attributes exist for all classes.**

  `__doc__` : Variable storing the documentation string for that class.

  `__class__` : Variable which gives you a reference to the class from any instance of it.

  `__module__` : Variable which gives you a reference to the module in which the particular class is defined.

  `__dict__` :The dictionary that is actually the namespace for a class (but not its superclasses).

- **Useful:**

  - `dir(x)` **returns a list of all methods and attributes defined for object x**

Special Data Items – Example

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

```
>>> print f.__doc__
A class representing a student.


>>> f.__class__
< class studentClass at 010B4C6 >


>>> g = f.__class__("Tom Jones", 34)


```

File Processing, Error Handling
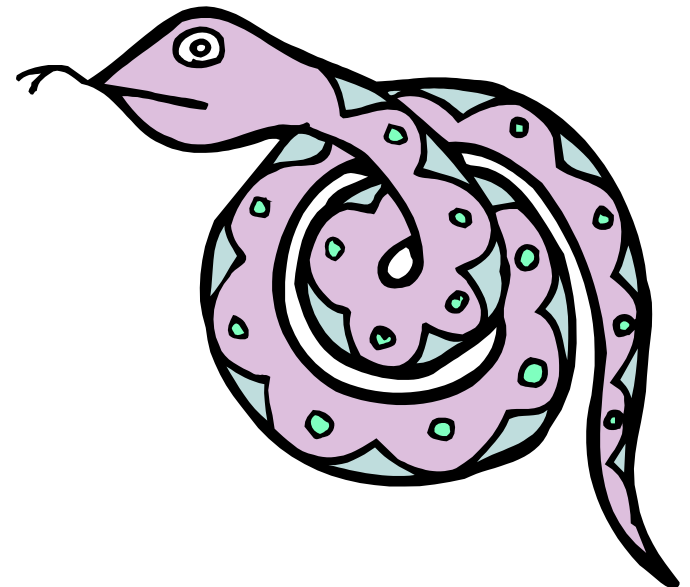
File Processing with Python

About what you'd expect….

```
fileptr = open("filename")
somestring = fileptr.read()
for line in fileptr:
    print line
fileptr.close()
```

Exception Handling

- **Exceptions are Python objects**
    - More specific kinds of errors are subclasses of the general Error class.

- **You use the following forms to interact with them:**
    - try
    - except
    - *else*
    - finally

for example...

```
>>> def divide(x, y):
        try:
            result = x / y
        except ZeroDivisionError:
            print "division by zero!"
        else:
            print "result is", result
        finally:
            print "executing finally clause"

>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Iterators

Iterators in Python

```
>>> for e in [1,2]:
     print e
1
2
>>> s = [1,2]
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
(most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Class with iterator in Python

```python
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
    def __iter__(self):
        return self
>>> for char in Reverse('spam'):
        print char
m
a
p
s
```

An iterator is any object with a "next" method

Iterators and list comprehensions

```
>>> [x for x in Reverse('spam')]

['m', 'a', 'p', 's']

>>> [x + x for x in Reverse('spam')]

['mm', 'aa', 'pp', 'ss']
```

Generators
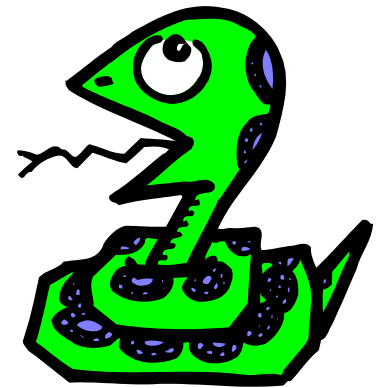
Generators

- **Defines an iterator with a function**
- **Maintains local state automatically**

```
def reverse(data):
    for i in range(len(data)):
        yield data[len(data)-1-i]

>>> for char in reverse('spam'):
        print char
m
a
p
s
```

Using generators

- **Merging sequences:**

```python
def merge(l, r):
    llen, rlen, i, j = len(l), len(r), 0, 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```

Using generators

```
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
        print g.next()
1
2
3
4
5


Traceback (most recent call last):
  File "<pyshell#73>", line 2, in <module>
    print g.next()
StopIteration

>>> [x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```

Generators and exceptions

```
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
        try:
          print g.next()
        except StopIteration:
          print 'Done'
          break
1
2
3
4
5
Done
```

List *Generators*

```
>>> a = (x * x for x in xrange(5))

>>> a
>>> <generator object <genexpr> at 0x031A7A80>

>>> for x in a:
   ...:      print x
   ...:
0
1
4
9
16
```

A directed graph class

```
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> print d
1 -> 2
1 -> 3
2 -> 4
4 -> 3
4 -> 1
```
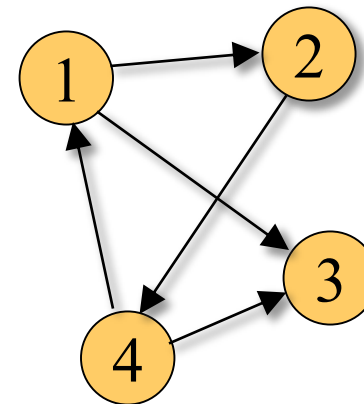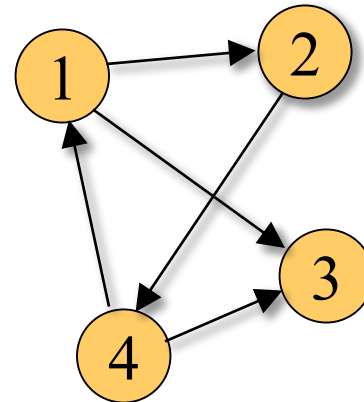
A directed graph class

```
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> [v for v in d.search(1)]
[1, 2, 4, 3]
>>> [v for v in d.search(4)]
[4, 3, 1, 2]
>>> [v for v in d.search(2)]
[2, 4, 3, 1]
>>> [v for v in d.search(3)]
[3]
```
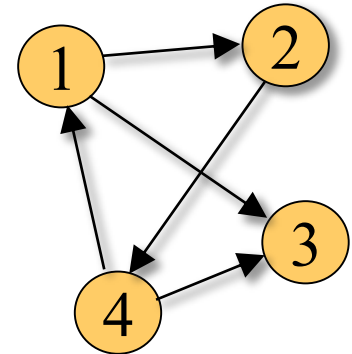

`search` method returns a *generator* for the nodes that can be reached from a given node by following arrows "from tail to head"

The DiGraph constructor

```
class DiGraph:

  def __init__(self, edges):
    self.adj = {}
    for u,v in edges:
        if u not in self.adj: self.adj[u] = [v]
        else: self.adj[u].append(v)

  def __str__(self):
    return '\n'.join(['%s -> %s'%(u,v) \
                      for u in self.adj for v in self.adj[u]])
  ...
```
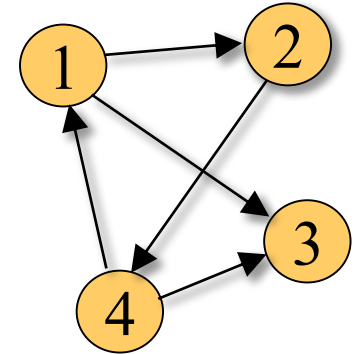
```
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> [v for v in d.search(1)]
{1: [2, 3], 2: [4], 4: [3, 1]}
```

The constructor builds a dictionary (`self.adj`) mapping each node name to a list of node names that can be reached by following one edge (an "adjacency list")

The search method

```python
class DiGraph:

    ...

    def search(self, u, visited=set()):
        # If we haven't already visited this node...
        if u not in visited:
            # yield it
            yield u
            # and remember we've visited it now.
            visited.add(u)
            # Then, if there are any adjacant nodes...
            if u in self.adj:
                # for each adjacent node...
                for v in self.adj[u]:
                    # search for all nodes reachable from *it*...
                    for w in self.search(v, visited):
                        # and yield each one.
                        yield w
```