Learning Python

A course in programming

Per Kraulis

9, 10 and 15 March 2004

Course literature

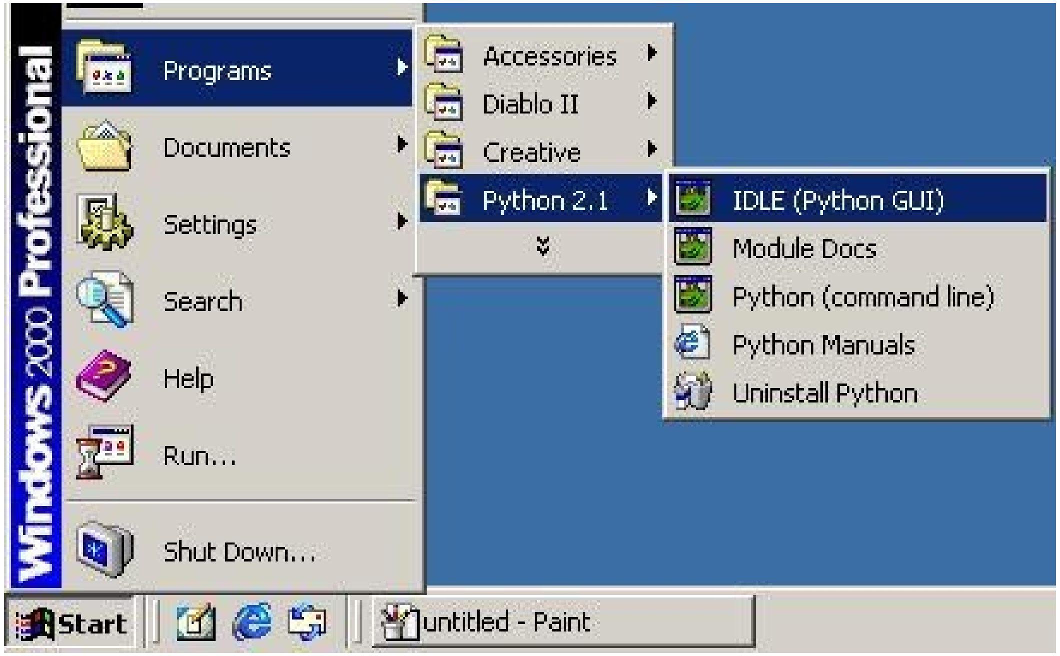
- Learning Python (Lutz & Ascher)
  - Tutorial
  - Covers Python 2.3
  - Should be read in sequence, but skip most of Part 1
  - Many useful points about sensible style
- Python in a Nutshell (Martelli)
  - Quick reference
  - Covers Python 2.2
  - Useful for looking things up things; use the index
  - Companion to the on-line documentation
  - Should be read in sequence only by serious nerds

Python under Windows

- Use IDLE to edit and run script files
- Or, use emacs to edit and run script files (setup required)
- Or, use the Python command line window (for interactive work)
- Or, double-click an existing Python file to run it
  - However, if results print to 'standard out', then problematic
  - Use the ‘raw_input’ trick (ugly)
  - Not a problem if graphics output

Let’s use IDLE in this course

- Graphical User Interface (GUI)
- Interactive Development Environment (IDE) for Python
- Useful under Windows and Unix
- Not required; use another editor if you want to
- Written in Python, using Tk
- Named for Eric Idle, member of Monty Python



IDLE startup message

```
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1200 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0.2
>>>
```

- Version number shown
- Interactive mode:
  - Awaits commands from the user
  - Commands must be valid Python code
  - Useful for tests and messing about
- Try the commands:
  - copyright
  - credits
  - license()

Part 1:
Variables and built-in types

Hello world!

```
>>> print 'Hello world!'
Hello world!
```

```
>>> a = 'Hello world!'
>>> print a
Hello world!
```

Variables and types

```
>>> a = 'Hello world!'    # this is an assignment statement
>>> print a
'Hello world!'
>>> type(a)               # expression: outputs the value in interactive mode
<type 'str'>
```

- Variables are created when they are assigned
- No declaration required
- The variable name is case sensitive: ‘val’ is not the same as ‘Val’
- The type of the variable is determined by Python
- A variable can be reassigned to whatever, whenever

```
>>> n = 12
>>> print n
12
>>> type(n)
<type 'int'>

>>> n = 12.0
>>> type(n)
<type 'float'>
```

```
>>> n = 'apa'
>>> print n
'apa'
>>> type(n)
<type 'str'>
```

Numbers

- Integers: 12 0 -12987 0123 0X1A2
  - Type ‘int’
  - Can’t be larger than 2**31
  - Octal literals begin with 0 (0981 illegal!)
  - Hex literals begin with 0X, contain 0-9 and A-F
- Floating point: 12.03 1E1 -1.54E-21
  - Type ‘float’
  - Same precision and magnitude as C double
- Long integers: 10294L
  - Type ‘long’
  - Any magnitude
  - Python usually handles conversions from int to long
- Complex numbers: 1+3J
  - Type ‘complex’

Numeric expressions

- The usual numeric expression operators: +, -, /, *, **, %, //
- Precedence and parentheses work as expected

```
>>> 12+5
17
>>> 12+5*2
22
>>> (12+5)*2
34
```

```
>>> 4 + 5.5
9.5
>>> 1 + 3.0**2
10.0
>>> 1+2j + 3-4j
(4-2j)
```

- Full list on page 58 in ‘Learning Python’

```
>>> a=12+5
>>> print a
17
>>> b = 12.4 + a             # 'a' converted to float automatically
>>> b                        # uses function 'repr'
29.399999999999999
>>> print b                  # uses function 'str'
29.4
```

Boolean expressions

- ‘True’ and ‘ False’ are predefined values; actually integers 1 and 0
- Value 0 is considered False, all other values True
- The usual Boolean expression operators: not, and, or

```
>>> True or False
True
>>> not ((True and False) or True)
False
>>> True * 12
12
>>> 0 and 1
0
```

- Comparison operators produce Boolean values
- The usual suspects: <, <=, >, >=, ==, !=

```
>>> 12<13
True
>>> 12>13
False
>>> 12<=12
True
>>> 12!=13
True
```

String

```
>>> a = 'Hello world!'
>>> b = "Hello world!"
>>> a == b
True
```

```
>>> a = "Per's lecture"
>>> print a
Per's lecture
```

- Single quotes or double quotes can be used for string literals
- Produces exactly the same value
- Special characters in string literals: \n newline, \t tab, others
- Triple quotes useful for large chunks of text in program code

```
>>> a = "One line.\nAnother line."
>>> print a
One line.
Another line.
```

```
>>> b = """One line,
another line."""
>>> print b
One line,
another line.
```

String conversions

```
>>> a = "58"
>>> type(a)
<type 'str'>
>>> b=int(a)
>>> b
58
>>> type(b)
<type 'int'>
```

```
>>> f = float('1.2e-3')
>>> f                       # uses 'repr'
0.0011999999999999999
>>> print f                 # uses 'str'
0.0012
>>> eval('23-12')
11
```

- Convert data types using functions ‘str’, ‘int’, ‘float’
- ‘repr’ is a variant of ‘str’
  - intended for strict, code-like representation of values
  - ‘str’ usually gives nicer-looking representation
- Function ‘eval’ interprets a string as a Python expression

```
>>> c = int('blah')            # what happens when something illegal is done?
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in -toplevel-
    c = int('blah')
ValueError: invalid literal for int(): blah
```

String operations

- Common string operations on page 75 in ‘Learning Python’

```
>>> a = "Part 1"
>>> b = "and part 2"
>>> a + ' ' + b           # concatenation, adding strings
'Part 1 and part 2'
>>> s = a * 2             # repeat and concatenate string
>>> print s
Part 1Part 1
```

```
>>> s[0]                  # index: one single character, offset 0 (zero)
'P'
>>> s[0:4]                # slice: part of string
'Part‘
>>> s[5:]                 # leave out one boundary: to the end
'1Part 1'
>>> >>> s[6:-1]           # negative index counts from the end
'Part '
```

```
>>> len(s)                # function ‘len’ to get length of string
12
>>> 'p' in s              # membership test
False
>>> 'P' in s
True
>>> 'Part' in s           # also works for substrings (new feature)
True
```

Changing strings. Not!

```
>>> s[0] = 'B'
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in -toplevel-
    s[0] = 'B'
TypeError: object doesn't support item assignment
```

- A string cannot be changed in Python! Immutable
- Good reasons for this; more later
- Create new strings from bits and pieces of old

```
>>> s = 'B' + s[1:]
>>> s
'Bart 1Part 1'
```

- Recreating strings may use a lot of computing power
- If you need to create many new strings, learn string formatting (more later)
- List processing can often be used to make string handling more efficient

String methods

- Strings have a set of built-in methods
- No method ever changes the original string!
- Several methods produce new strings
- A list on page 91 in ‘Learning Python’

```
>>> s = 'a string, with stuff'
>>> s.count('st')                  # how many substrings?
2
>>> s.find('stu')                  # give location of substring, if any
15
>>> three = '3'
>>> three.isdigit()                # only digit characters in string?
True
```

```
>>> supper = s.upper()             # convert to upper case
>>> supper
'A STRING, WITH STUFF'
>>> s.rjust(30)                    # right justify by adding blanks
'          a string, with stuff'
>>> "newlines\n\n\n".strip()       # a string literal also has methods!
'newlines'
```

```
>>> s.replace('stuff', 'characters') # replace substring (all occurrences)
'a string, with characters'
>>> s.replace('s', 'X', 1)         # replace only once
'a Xtring, with stuff'
```

List

- Ordered collection of objects; array
- Heterogenous; may contain mix of objects of any type

```
>>> r = [1, 2.0, 3, 5]       # list literal; different types of values
>>> r
[1, 2.0, 3, 5]
>>> type(r)
<type 'list'>
```

```
>>> r[1]                     # access by index; offset 0 (zero)
2.0
>>> r[-1]                    # negative index counts from end
5
```

```
>>> r[1:3]                   # a slice out of a list; gives another list
[2.0, 3]
```

```
>>> w = r + [10, 19]         # concatenate lists; gives another list
>>> w
[1, 2.0, 3, 5, 10, 19]
>>> r                        # original list unchanged; w and r are different
[1, 2.0, 3, 5]
```

```
>>> t = [0.0] * 10           # create an initial vector using repetition
>>> t
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

List operations

- Lists are mutable; can be changed in-place
- Lists are dynamic; size may be changed

```
>>> r = [1, 2.0, 3, 5]
>>> r[3] = 'word'            # replace an item by index
>>> r
[1, 2.0, 3, 'word']
```

```
>>> r[0] = [9, 8]            # lists can be nested
>>> r
[[9, 8], 2.0, 3, 'word']
```

```
>>> r[0:3] = [1, 2, 5, 6]    # change a slice of list; may change list length
>>> r
[1, 2, 5, 6, 'word']
>>> r[1:3] = []              # remove items by setting slice to empty list
>>> r
[1, 6, 'word']
```

```
>>> len(r)                   # length of list; number of items
3
```

```
>>> 6 in r                   # membership test
True
>>> r.index(6)               # search for position; bombs if item not in list
1
```

List methods, part 1

- Lists have a set of built-in methods
- Some methods change the list in-place

```
>>> r = [1, 2.0, 3, 5]
>>> r.append('thing')              # add a single item to the end
>>> r
[1, 2.0, 3, 5, 'thing']
>>> r.append(['another', 'list'])  # list treated as a single item
>>> r
[1, 2.0, 3, 5, 'thing', ['another', 'list']]
```

```
>>> r = [1, 2.0, 3, 5]
>>> r.extend(['item', 'another'])  # list items appended one by one
>>> r
[1, 2.0, 3, 5, 'item', 'another']
```

```
>>> k = r.pop()                    # remove last item from list and return
>>> k
'another'
>>> r
[1, 2.0, 3, 5, 'item']
```

- Methods 'append' and 'pop' can be used to implement a stack

List methods, part 2

- Use the built-in 'sort' method: efficient
- The list is sorted in-place; a new list is not produced!

```
>>> r = [2, 5, -1, 0, 20]
>>> r.sort()
>>> r
[-1, 0, 2, 5, 20]
```

```
>>> w = ['apa', '1', '2', '1234']
>>> w.sort()                       # strings: lexical sort using ASCII order
>>> w
['1', '1234', '2', 'apa']
```

```
>>> w.reverse()                    # how to flip a list; in-place!
>>> w
['apa', '2', '1234', '1']
```

```
>>> v = w[:]                       # first create a copy of the list
>>> v.reverse()                    # then reverse the copy
>>> v                              # use same technique for sort
['1', '1234', '2', 'apa']
>>> w
['apa', '2', '1234', '1']
```

Converting lists between strings

```
>>> s = 'biovitrum'                            # create a string
>>> w = list(s)                                # convert into a list of
char's
>>> w
['b', 'i', 'o', 'v', 'i', 't', 'r', 'u', 'm']
>>> w.reverse()
>>> w
['m', 'u', 'r', 't', 'i', 'v', 'o', 'i', 'b']
>>> r = ''.join(w)                             # join using empty string
>>> r
'murtivoib'
>>> d = '-'.join(w)                            # join using dash char
>>> d
'm-u-r-t-i-v-o-i-b'

>>> s = 'a few words'
>>> w = s.split()               # splits at white-space (blank, newline)
>>> w
['a', 'few', 'words']
```

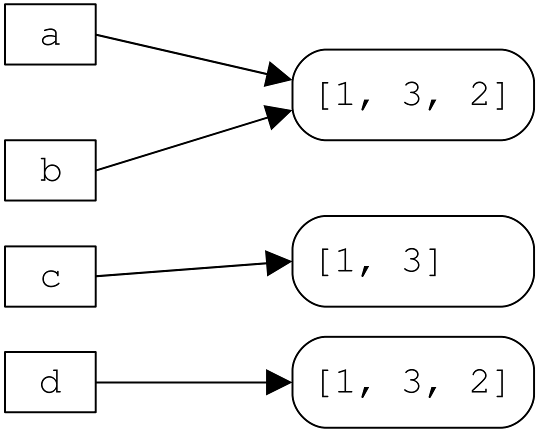
- 'split' is useful for simple parsing
- Otherwise use regular expression module 're'; later

```
>>> ' | '.join(w)             # use any string with method 'join'
'a | few | words'
```

Objects, names and references

- All values are objects
- A variable is a name referencing an object
- An object may have several names referencing it
- Important when modifying objects in-place!
- You may have to make proper copies to get the effect you want
- For immutable objects (numbers, strings), this is never a problem

```
>>> a = [1, 3, 2]
>>> b = a
>>> c = b[0:2]
>>> d = b[:]
```



```
>>> b.sort()     # 'a' is affected!
>>> a
[1, 2, 3]
```

Dictionary

- An unordered collection of key/value pairs
- Each key maps to a value
- Also called "mapping", "hash table" or "lookup table"

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> h['key']                                     # access by key
12
>>> h.has_key('nyckel')
True
```

```
>>> h['Per'] = 'Kraulis'                         # adding a key/value
>>> h
{'nyckel': 'word', 'Per': 'Kraulis', 'key': 12}  # the output order is random
>>> h['Per'] = 'Johansson'                       # replaces the value
>>> h
{'nyckel': 'word', 'Per': 'Johansson', 'key': 12}
```

- The key is
  - Usually an integer or a string
  - Should (must!) be an immutable object
  - May be any object that is 'hashable' (more later)
  - Any key occurs at most once in a dictionary!
- The value may be any object
  - Values may occur many times

Forgetting things: 'del'

- Use command 'del' to get rid of stuff
- Command! Not function!
- Actually removes variables (names), not objects

```
>>> a = 'thing'                                       # define a variable
>>> a
'thing'
>>> del a                                             # forget about the variable
>>> a
Traceback (most recent call last):
  File "<pyshell#182>", line 1, in -toplevel-
    a
NameError: name 'a' is not defined
```

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> del h['key']                                 # remove the key and its value
>>> h
{'nyckel': 'word'}
```

```
>>> r = [1, 3, 2]
>>> del r[1]                                     # another way of removing list
items
>>> r
[1, 2]
```

Forgetting things: garbage collection

- What happens to the object when its name is ’del’ed, or reassigned to another object?

- Don’t worry, be happy!

- The Python systems detects when an object is ’lost in space’
  - It keeps track of the number of references to it

- The object’s memory gets reclaimed; garbage collection

- A few problematic special cases; cyclical references

Dictionary methods, part 1

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> 'Per' in h                       # test if key in dictionary
False
>>> h['Per']
Traceback (most recent call last):
  File "<pyshell#192>", line 1, in -toplevel-
    h['Per']
KeyError: 'Per'
```

```
>>> h.get('Per', 'unknown')          # return value, or default if not found
'unknown'
>>> h.get('key', 'unknown')
12
```

```
>>> h.keys()                 # all keys in a list; unordered
['nyckel', 'key']
>>> h.values()               # all values in a list; unordered
['word', 12]
```

```
>>> len(h)                   # number of keys in dictionary
2
```

Dictionary methods, part 2

```
>>> g = h.copy()             # a separate copy of the dictionary
>>> del h['key']
>>> h
{'nyckel': 'word'}
>>> g
{'nyckel': 'word', 'key': 12}
```

```
>>> h['Per'] = 'Johansson'
>>> h
{'nyckel': 'word', 'Per': 'Johansson'}
>>> h.update(g)                          # add or update all key/value from g
>>> h
{'nyckel': 'word', 'key': 12, 'Per': 'Johansson'}
```

Tuple

- Same as list, except immutable
- Once created, can't be changed
- Some functions return tuples

```
>>> t = (1, 3, 2)
>>> t[1]                          # access by index; offset 0 (zero)
3
```

```
>>> (a, b, c) = t                 # tuple assignment (unpacking)
>>> a
1
>>> b
3
>>> a, b, c                       # actually a tuple expression!
(1, 3, 2)
```

```
>>> a, b = b, a                   # neat trick to swap values
>>> a, b
(3, 1)
```

```
>>> r = list(t)                   # convert tuple to a list
>>> r
[1, 3, 2]
>>> tuple(r)                      # convert list to a tuple
(1, 3, 2)
```

String formatting

- String formatting operator '%'
- Usually the best way to create new strings
- C-like formatting: Slightly tricky, but powerful
- Many string formatting codes
  - %s: string (uses function 'str')
  - %r: string (uses function 'repr')
  - %f, %e, %g: float

```
>>> w = "Number %i won!" % 12     # string formatting operator %
>>> w
'Number 12 won!'
```

- Tuples are used as operands in string formatting when >1 items
- The length of the tuple must match the number of format codes in the string
- Lists won't do!

```
>>> c = 'Python'
>>> n = 11
>>> "This is a %s course with %i students." % (c, n)
'This is a Python course with 11 students.'
```

Part 2:
Statements

Write scripts in IDLE

- Now we need to write proper scripts, saved in files
- In IDLE:
  - 'File'
  - 'New Window'
  - Do immediately 'Save as…'
    - Browse to directory 'Desktop'
    - Create a directory 'Python course'
    - Go down into it
    - Enter the file name 't1.py'
    - Save
- Work in the window called 't1.py'
  - Enter the following code:

```
"file t1.py"                # this is a documentation string
print "Hello world!"
```

  - Save the file: Ctrl-S, or menu 'File', 'Save'
  - Run the script: F5, or menu 'Run', 'Run Module'

'if' statement; block structure

- The Python feature that one either loves or hates
- Block structure is determined by indentation

- Edit a new script file 't2.py'
  - In window 't1.py' do 'File', 'New Window', then 'Save As…
- Use the 'if' statement:

```
"file t2.py"

person = 'Luke'

if person == 'Per':
    status = 'Pythonist'
elif person == 'Luke':
    status = 'Jedi knight'
else:
    status = 'unknown'

print person, status
```

- Note that the IDLE editor helps with indentation
- Run the script (F5)

Dictionary often better than if… elif…

- Particularly with many hardcoded choices (elif's)…

```
"file t3.py"

status_map = {'Luke': 'Jedi Knight',
              'Per': 'Pythonist'}

person = 'Luke'

print person, status_map.get(person, 'unknown')
```

- More compact, and more efficient
- This pattern is very useful

Built-in types and their Boolean interpretations

int	0	False
	-1	True
	124	True
float	0.0	False
str	""	False
	"False"	True !
dict	{ }	False
	{'key': 'val'}	True
list	[ ]	False
	[False]	True !

- All built-in types can be used directly in 'if' statements

- Zero-valued numbers are False
- All other numbers are True

- Empty containers (str, list, dict) are False
- All other container values are True

- Use function 'bool' to get explicit value

'for' statement

- Repetition of a block of statements
- Iterate through a sequence (list, tuple, string, iterator)

```
"file t4.py"

s = 0
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8]:     # walk through list, assign to i
    s = s + i
    if s > 10:
        break                             # quit 'for' loop, jump to after it

print "i=%i, s=%i" % (i, s)
```

```
"file t5.py"

r = []
for c in 'this is a string with blanks':   # walks through string, char by
char
    if c == ' ': continue                 # skip rest of block, continue loop
    r.append(c)

print ''.join(r)
```

Built-in functions 'range' and 'xrange'

- Built-in functions 'range' and 'xrange' useful with 'for'
- 'range' creates a list
- Warning: may use lots of memory; inefficient!

```
>>> range(9)                     # start=0, step=1 by default
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> range(1, 12, 3)              # explicit start, end, step
[1, 4, 7, 10]
>>> range(10**9)                 # MemoryError!
```

- 'xrange' creates an iterator, which works like a list
- Very memory-efficient!

```
"file t6.py"

s = 0
for i in xrange(100000):
    if i % 19 == 0:              # remainder: evenly divisible with 19?
        s = s + i

print s
```

'while' statement

- Repetition of a block of statements
- Loop until test becomes false, or 'break'

```
"file t7.py"

r = []
n = 0
last = 20

while n <= last:           # any expression interpretable as Boolean
    r.append(str(n))
    n += 3

print ', '.join(r)
```

Optional 'else' block in loops

- 'else' block executed if no 'break' encountered
- May often replace success/failure flags
- Valid in 'for' and 'while' loops

```
"file t8.py"

r = [1, 3, 10, 98, -2, 48]

for i in r:
    if i < 0:
        print 'input contains negative value!'
        break                   # this skips out of loop, including 'else'
    else:
        pass                    # do-nothing statement
else:                           # do if loop ended normally
    print 'input is OK'
```

- 'pass' statement does absolutely nothing
- May be useful as placeholder for unwritten code
- A few cases where required (more later)

Error handling: ’try’ and ’except’

- Run-time error normally causes execution to bomb
- The error message gives type of error
- Use a ’try’, ’except’ blocks to catch and handle errors

```
"file t9.py"

numbers = []
not_numbers = []

for s in ['12', '-4.1', '1.0e2', 'e3']:
    try:
        n = float(s)
        numbers.append(s)
    except ValueError, msg:
        not_numbers.append(str(msg))

print 'numbers:', numbers
print 'not numbers:', not_numbers
```

```
numbers: ['12', '-4.1', '1.0e2']
not numbers: ['invalid literal for float(): e3']
```

How to split up long lines

- Sometimes a source code line needs splitting up
- Indentation rule means we do not have free-format!

```
"illegal syntax example"

if a_complicated_expression and
   another_complicated_expression:
    print 'this is illegal syntax; it will not work'
```

- Alt 1: Use continuation character '\' as the very last

```
"valid syntax example 1"

if a_complicated_expression and \
    another_complicated_expression:
    print 'this is valid syntax'
```

- Alt 2: Enclose the expression in parenthesis
  - Lines within parenthesis can be broken up
  - Also true for [ ] and { }

```
"valid syntax example 2"

if (a_complicated_expression and
    another_complicated_expression):
    print 'this is valid syntax'
```

Statements not covered in this course

- 'finally': used with 'try', 'except'
- 'raise': causes an exception
- 'yield': in functions
- 'global': within functions
- 'exec': execute strings as code

- **There is no 'goto' statement!**

Interlude:
About Python

What is Python?

- A programming language
  - Features from Perl and Java, with influences from C, C++, Scheme, Haskell, Smalltalk, Ada, Simula,…
- Open Source
  - Free; source code available
  - Download on your own system
- Written by Guido van Rossum
- Monty Python's Flying Circus…

- First release Feb 1991: 0.9.0
- Current version: 2.3
- Evolutionary policy for changes between versions
- Backward incompatibility issues are rare
  - But they do exist…
  - Largest change: from 1.5 to 2.0

Features of Python

- A script language
- Interpreted
  - No compile/link stage
  - Write and run
  - Slow compared to C, C++
- Elegant design; “tight”
- Designed for
  - Quick-and-dirty scripts
  - Reusable modules
  - Very large systems
- Object-oriented
  - Very well-designed
  - But you don’t have to use it
- Useful error messages
- Automatic memory handling
- Independent of operating system
  - Works on both Unix and Windows
  - Even file system code can be made os-independent
- Large library of standard modules
- Large number of third-party modules
- Integration with external C code
- Well documented

Part 3:
Functions

How to define your own function

- Use the 'def' statement
- Function body follows; indented!
- This is a statement like others
  - Can be placed basically anywhere
  - Required: Define the function before calling it

```
"file t10.py"

def all_items_positive(r):                           # function definition
    "Check that all values in list r are positive."  # documentation string
    for i in r:
        if i <= 0: return False                      # return a result value
    return True

sequences = [[1, 5, 6, -0.01], [0.01, 1, 2]]

for seq in sequences:
    if not all_items_positive(seq):                  # call the function
        print 'invalid: ', seq
```

Function features

- The value of an argument is not checked for type
  - Often very useful; overloading without effort
  - Of course, the function may still bomb if invalid value is given

- The documentation string is <u>not</u> required (more later)
  - But strongly encouraged!
  - Make a habit of writing one (before writing the function code)

- A user-defined function has exactly the same status as a built-in function, or a function from another module

Function arguments: fixed

- Fixed number of arguments
- Associated by order

```
"file t11.py"

def fixed_args(a, c, b):                            # note!: order of args
    "Format arguments into a string and return.“    # doc string
    return "a=%s, b=%s, c=%s" % (a, b, c)           # '%s' converts to string

print fixed_args('stuff', 1.2, [2, 1])
```

Function arguments: variable

- List of any number of arguments
- Useful when unknown number of arguments needed
- The argument values collected into a tuple
  - Called 'args', by convention
  - The ’*’ is the magical part

```
"file t12.py"

def unspec_args(a, *args):                 # name 'args' is a convention
    return "a=%s, others=%s" % (a, args)

print unspec_args('bla', 'qwe', 23, False)
```

```
a=bla, others=('qwe', 23, False)
```

Function arguments: default values

- Arguments may have default values
- When argument not given in a call, default value is used
- If no default value, and not given when called: bombs
- Use explicit names to override argument order

```
"file t13.py"

def default_args(a, b='bar', c=13):
    return "a=%s, b=%s, c=%s" % (a, b, c)

print default_args('apa')                   # uses all default values
print default_args('s', b='py')             # overrides one default value
print default_args(c=-26, a='apa')          # override argument order
```

```
a=apa, b=bar, c=13
a=s, b=py, c=13
a=apa, b=bar, c=-26
```

Function arguments: keywords

- Keyword/value arguments
- The argument values collected into a dictionary
  - Called 'kwargs', by convention
  - The ’**’ is the magical part
  - First attempts to match existing argument names

```
"file t14.py"

def keyword_args(a, b='bla', **kwargs):
    return "a=%s, b=%s, kwargs=%s" % (a, b, str(kwargs))

print keyword_args('stuff', c='call')
print keyword_args('stuff', c='call', b='apa')
print keyword_args(c='call', d=12, a='gr')
```

```
a=stuff, b=bla, kwargs={'c': 'call'}
a=stuff, b=apa, kwargs={'c': 'call'}
a=gr, b=bla, kwargs={'c': 'call', 'd': 12}
```

Function arguments: explicit type checking

- Use the 'assert' statement
- Checks that its Boolean expression is True, else bombs
- Can be used for sanity checks anywhere in code
- Optional explanatory message (or data)

```
"file t15.py"

def fixed_args(a, c, b):
    assert type(a) == type(1), "'a' must be an integer"
    return "a=%s, b=%s, c=%s" % (a, b, c)

print fixed_args('a', 1.2, [2, 1])
```

```
Traceback (most recent call last):
  File "C:\Python tests\t15.py", line 8, in -toplevel-
    print fixed_args('a', 1.2, [2, 1])
  File "C:\Python tests\t15.py", line 5, in fixed_args
    assert type(a) == type(1), "'a' must be an integer"
AssertionError: 'a' must be an integer
```

Function arguments: local variables

- Arguments become local variables
  - Immutable values are copied, in effect
  - Mutable values may still be changed: be careful
- Variables created within 'def' block are local
  - Forgotten on return

```
"file t16.py"

def test_local(a, r):
    print 'local original ', a, r
    a = 12
    r[1] = 999
    print 'local changed  ', a, r

a = -5
r = [0, 1, 2]
print 'global original', a, r
test_local(a, r)
print 'global changed ', a, r
```

```
global original -5 [0, 1, 2]
local original  -5 [0, 1, 2]
local changed   12 [0, 999, 2]
global changed  -5 [0, 999, 2]
```

Function without 'return': value None

- A function does not have to use the 'return' statement
- If not, then same as a 'procedure' in other languages
- Actually returns a value anyway: 'None'
- A 'return' without value is OK: returns 'None'
- 'None' is a special value meaning 'nothing'
  - Useful in many contexts
  - Particularly in object-oriented programming (more later)

```
"file t17.py"

def concat_strings(a, b):
    str_type = type('')                             # save a type value!
    if type(a) == str_type and type(b) == str_type:
        return a + ' ' + b

print 'strings:', concat_strings('first', 'second')
print 'integers:', concat_strings(1, 2)
```

```
strings: first second
integers: None
```

The 'math' module: functions and constants

- A peek at modules
- Math functions available in a separate module

```
"file t18.py"

from math import *             # import everything from module 'math'

print e, pi
print cos(radians(180.0))
print log(10.0)
print exp(-1.0)
```

```
2.71828182846 3.14159265359
-1.0
2.30258509299
0.367879441171
```

Functions are objects; names are references

- A function is just another kind of object
- Nothing magical about their names; can be changed

```
"file t19.py"

from math import *

def print_calc(f):
    print "log(%s)=%s, exp(%s)=%s" % (f, log(f), f, exp(f))

print_calc(1.0)
log, exp = exp, log       # evil code! swap the objects the names refer to
print_calc(1.0)
```

```
log(1.0)=0.0, exp(1.0)=2.71828182846
log(1.0)=2.71828182846, exp(1.0)=0.0
```

- A function can be passed as any argument to another function
- A function can assigned to a variable

Built-in function 'map'

- Built-in function that works on a list
- 'map' takes a function and a list
  - The function must take only one argument, and return one value
  - The function is applied to each value of the list
  - The resulting values are returned in a list

```
>>> from math import *
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> map(cos, r)
[1.0, 0.54030230586813977, -0.41614683654714241, -0.98999249660044542,
 -0.65364362086361194, 0.28366218546322625, 0.96017028665036597]
```

Built-in function 'reduce'

- Built-in function that works on a list
- 'reduce' takes a function and a list
- It boils down the list into one value using the function
  - The function must take only two arguments, and return one value
  - 'reduce' applies the function to the first two values in the list
  - The function is then applied to the result and the next value in the list
  - And so on, until all values in the list have been used

```
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> def sum(x, y): return x+y

>>> reduce(sum, r)                # (((((1+2)+3)+4)+5)+6)
21
```

Built-in function 'filter'

- Built-in function that works on a list
- 'filter' takes a function and a list
- It uses the function to decide which values to put into the resulting list
  - Each value in the list is given to the function
  - If the function return True, then the value is put into the resulting list
  - If the function returns False, then the value is skipped

```
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> def large(x): return x>3

>>> filter(large, r)
[4, 5, 6]
```

Files: reading

- A file object is created by the built-in function 'open'
- The file object has a set of methods
- The 'read' methods get data sequentially from the file
  - 'read': Get the entire file (or N bytes) and return as a single string
  - 'readline': Read a line (up to and including newline)
  - 'readlines': Read all lines and return as a list of strings

```
>>> f = open('test.txt')      # by default: read-only mode
>>> line = f.readline()       # read a single line
>>> line
'This is the first line.\n'
>>> lines = f.readlines()     # read all remaining lines
>>> lines
['This is the second.\n', 'And third.\n']
```

- Several modules define objects that are file-like, i.e. have methods that make them behave as files

Files: writing

- The 'write' method simply outputs the given string
- The string does not have to be ASCII; binary contents allowed

```
>>> w = open('output.txt', 'w')         # write mode (text by default)
>>> w.write('stuff')                    # does *not* add newline
automatically
>>> w.write('\n')
>>> w.write('more\n and even more\n')
>>> w.close()
```

```
stuff
more
 and even more
```

Files: read by 'for' loop

- Iteration using the 'for' loop over the file reads line by line
- The preferred way to do it

```
"file t20.py"

infile = open('test.txt')               # read-only mode
outfile = open('test_upper.txt', 'w')   # write mode; creates the file

for line in infile:
    outfile.write(line.upper())

infile.close()                   # not strictly required; done automatically
outfile.close()
```

- Note: Each line will contain the trailing newline '\n' character
- Use string method 'strip' or 'rstrip' to get rid of it

Files, old-style read strategies

- Previous versions of Python did not have the 'for line in file' feature
- Instead, the following alternatives were used:

```
for line in infile.readlines():     # reads entire file into list of lines
    do_something(line)
```

```
for line in infile.xreadlines():    # like xrange: more memory-efficient
    do_something(line)
```

```
line = infile.readline()
while line:                         # line will be empty only at end-of-file
    do_something(line)
    line = infile.readline()
```

- The last alternative works because 'readline' returns the line including the final newline '\n' character
- Only when end-of-file is reached will a completely empty line be returned, which has the Boolean value 'False'

Part 4:
Modules

Example: Reverse complement NT sequence

- Given a nucleotide sequence, produce reverse complement
- Use available features

```
"file t21.py"

complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

seq = 'cgtaacggtcaggttatattt'

complist = map(complement_map.get, seq)
complist.reverse()
revseq = ''.join(complist)

print seq
print revseq
```

```
cgtaacggtcaggttatattt
aaatataacctgaccgttacg
```

Make the code more reusable

- How to make the example code more reusable?
- Step 1: Make a function

```
"file t22.py"

complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

def reverse_complement(seq):
    complist = map(complement_map.get, seq)
    complist.reverse()
    return ''.join(complist)

seq = 'cgtaacggtcaggttatattt'
print seq
print reverse_complement(seq)
```

Make a module of the code

- How to make the code even more reusable?
- Step 2: Make a module out of it
- Is actually already a module!
- Let’s simply rename it to ’ntseq.py’

```
"""file ntseq.py

Module 'ntseq': operations on NT sequences.
"""

complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

def reverse_complement(seq):
    "Return the reverse complement of an NT sequence."
    complist = map(complement_map.get, seq)
    complist.reverse()
    return ''.join(complist)

seq = 'cgtaacggtcaggttatattt'
print seq
print reverse_complement(seq)
```

How to use the module: ’import’ statement

- The ’import’ statement makes a module available
- The module name (not the file name) is imported: skip the ’.py’
- Access module features through the ’dot’ notation

```
"file t23.py"

import ntseq

seq = 'aaaccc'
print seq
print ntseq.reverse_complement(seq)
```

```
cgtaacggtcaggttatattt
aaatataacctgaccgttacg
aaaccc
gggttt
```

- Huh?! We got more than we wanted!
- First two lines: The test code in the module was also executed

Module self-test code: the ’__name__’ trick

- The ’import’ statement executes all code in the module file
- How to ’hide’ self-test code?
- Use the predefined variable ’__name__’:
  - If executed as the main program: value ’__main__’
  - If executed as a module: some other value

```
"""file ntseq_mod.py

Module 'ntseq_mod': operations on NT sequences.
"""

complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

def reverse_complement(seq):
    "Return the reverse complement of an NT sequence."
    complist = map(complement_map.get, seq)
    complist.reverse()
    return ''.join(complist)

if __name__ == '__main__':          # code to test the function
    seq = 'cgtaacggtcaggttatattt'
    print seq
    print reverse_complement(seq)
```

Now, the 'import' statement behaves

```
"file t24.py"

import ntseq_mod                         # note: ntseq_mod!

seq = 'aaaccc'
print seq
print ntseq_mod.reverse_complement(seq)  # note: ntseq_mod!
```

```
aaaccc
gggttt
```

How are modules found by ’import’?

- The ’import’ statement searches the directories named in sys.path
- The first file found ’xxx.py’ (where xxx is the module name) is used
- There are ways to change your sys.path according to your needs
  - Beyond this course; described in the book

```
"file t25.py"

import sys

for dirname in sys.path:
    print dirname
```

```
M:\My Documents\Python course\tests
C:\Python23\Lib\idlelib
C:\WINNT\system32\python23.zip
C:\Python23
C:\Python23\DLLs
C:\Python23\lib
C:\Python23\lib\plat-win
C:\Python23\lib\lib-tk
C:\Python23\lib\site-packages
```

Modules are easy, fun, and powerful

- The module feature is the basis for Python's ability to scale to really large software systems

- Easy to create: every Python source code file is a module!

- Features to make a module elegant:
  - Doc strings
  - '__name__' trick
  - Namespace concept

- Be sure to browse the standard library modules!
  - You will find extremely useful stuff there
  - You will learn good coding habits

- Packages are directories of several associated modules
  - Not covered in this course. A few minor interesting points

Namespaces

- A namespace is a bag of names
- A module is a namespace
- Use the built-in function ’dir’ to list the names in a namespace
- ’import’ statement modifies the namespace

```
"file t26.py"

for name in dir():
    print "%r: %r" % (name, eval(name))
print
print 'virgin namespace:', dir()

import ntseq_mod
print 'after import:', dir()

from ntseq_mod import *
print 'after from:', dir()
```

```
'__builtins__': <module '__builtin__' (built-in)>
'__doc__': 'file t26.py'
'__name__': '__main__'

virgin namespace: ['__builtins__', '__doc__', '__name__', 'name']
after import: ['__builtins__', '__doc__', '__name__', 'name', 'ntseq_mod']
after from: ['__builtins__', '__doc__', '__name__', 'complement_map', 'name',
             'ntseq_mod', 'reverse_complement']
```

Avoiding clutter in your namespace

- Using ’from module import *’ can create clutter
- Fine-tuning of import; bring in only selected things
- Rename things from a module
  - Avoid name collisions
  - Make it clearer

```
"file t27.py"

print 'virgin namespace:', dir()

from math import exp
from math import log as natural_logarithm

print 'selective import:', dir()
print 'exp(1.0)=', exp(1.0)
print 'natural_logarithm(10.0)=', natural_logarithm(10.0)
```

```
virgin namespace: ['__builtins__', '__doc__', '__name__']
selective import: ['__builtins__', '__doc__', '__name__',
                   'exp', 'natural_logarithm']
exp(1.0)= 2.71828182846
natural_logarithm(10.0)= 2.30258509299
```

The special ’__xxx__’ variables

- In Python, the variable names ’__xxx__’ are special
- Two underscores ’_’ at each end of a word
- Some are created automatically during execution
  - __name__, __file__, __doc__, __builtins__
- Some control the execution in different ways
- Only set the value of one when you know what you are doing!
- Don’t use this type of name for your own stuff!

Doc strings: ’__doc__’

- We have mentioned documentation strings before
  - The first string in a module
  - The first string after a ’def’ statement
- Accessed through variable ’__doc__’
- Feature to facilitate creation of documentation
  - Used by tools to produce documentation, such as ’pydoc’
  - See ’Module Docs’ in ’Start’ > ’Programs’ > ’Python 2.3’

```
>>> import ntseq_mod
>>> print ntseq_mod.__doc__
file ntseq_mod.py

Module 'ntseq': operations on NT sequences.

>>> print ntseq_mod.reverse_complement.__doc__
Return the reverse complement of an NT sequence.
```

Documentation resources

- The Python manual
  - ’Start’ > ’Programs’ > ’Python 2.3’ > ’Python Manuals’
  - Tutorial
  - Language Reference (heavy stuff)
  - Global Module Index (very useful)

- The Module Docs (pydoc)
  - Actually lists what’s on your system, including any installed 3rd party packages (if in the module path)
  - Uses the __doc__ strings in the module source code

- www.python.org
  - Pointers to other sites

- ”Use the source, Luke…”
  - Sometimes, the source code itself may be very useful

Python under Unix

```
% python
Python 2.3.3 (#51, Dec 18 2003, 20:22:39)…
Type "copyright", "credits" or "license()" for more information.
>>> print "Hello world!"
Hello world!
>>>
```

- Use any editor (emacs, vi,…) to edit and run script files
- Use IDLE to edit and run script files

```
% python script.py
…result output…
%
```

```
% emacs script.py         # edit the file…
% cat script.py
#!/usr/local/bin/python   # note the magical first line
print "Hello world!"
%
% chmod ugo+x script.py   # make the file executable
%
% script.py
Hello world!
%
```

Homework until the next lecture

1. Write a function to determine whether a sequence is AA, NT or something else.
    1. Test it thoroughly
    2. Make a module of it

2. Produce lists from the tab-delimited file ‘addresses.txt’.
    1. Sort by last name
    2. Sort by telephone number
    3. Sort by location
    4. Try your scripts using the file ‘addresses2.txt’. If it doesn’t work, fix it.

3. Write a simple parser for the FASTA format file ‘fasta.txt’.

Part 5:
Object-oriented programming, classes

Classes *vs.* objects (instances)

- A class is like a
  - Prototype
  - Blue-print ("ritning")
  - An object creator

- A class defines potential objects
  - What their structure will be
  - What they will be able to do

- Objects are instances of a class
  - An object is a container of data: attributes
  - An object has associated functions: methods

A class example: Geometrical shapes

- Let's define classes for geometrical shapes
  - With data; position, etc
  - With functions: compute area, etc

```
"file geom1.py: Module with classes for geometrical shapes, 1st try"

import math

class Circle:                           # class definition statement
    "A 2D circle."                      # documentation string

    def __init__(self, x, y, radius=1): # initialization method
        self.x = x                      # set the attributes of this instance
        self.y = y
        self.radius = radius

    def area(self):
        "Return the area of the shape."
        return math.pi * self.radius**2
```

Instances of classes

- Let's create some instances of the Circle class
- Look at attribute 'radius'
- Use the method 'area'

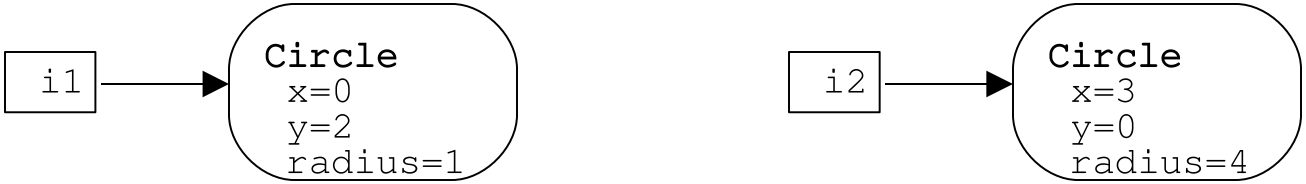
```
"file t28.py"

from geom1 import *

i1 = Circle(0, 2)              # '__init__' is called automatically
i2 = Circle(3, 0, 4)

print 'i1:', i1.radius, i1.area()
print 'i2:', i2.radius, i2.area()
print str(i1)
```

```
i1: 1 3.14159265359
i2: 4 50.2654824574
<geom1.Circle instance at 0x009CEA08>
```



Changing an instance: attribute assignment

- The values of attributes can be changed
- Simply assign the attribute a new value

```
"file t29.py"

from geom1 import *

i1 = Circle(0, 2)
print 'i1:', i1.radius, i1.area()

i1.radius = 2.5                      # change the value of an attribute
print 'i1:', i1.radius, i1.area()
```

```
i1: 1 3.14159265359
i1: 2.5 19.6349540849
```

Changing an instance: references

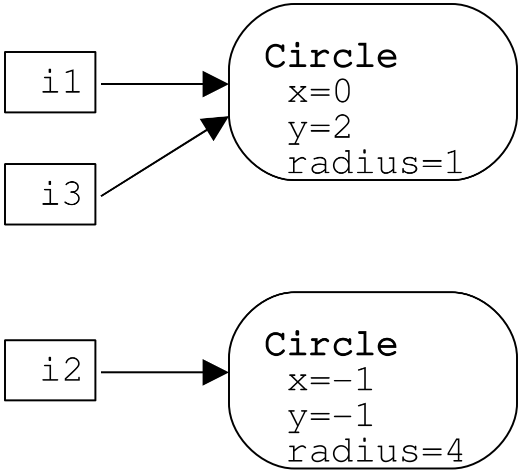
- Variables may reference the same object
- Changing an attribute changes the object, not the reference

```
"file t30.py"

from geom1 import *

i1 = Circle(0, 2)
i2 = Circle(-1, -1, 4)
i3 = i1

i1.radius = 1.75
print 'i1:', i1.radius
print 'i2:', i2.radius
print 'i3:', i3.radius
```



```
i1: 1.75
i2: 4
i3: 1.75
```

Changing an instance: attribute status

- Attributes are local to the instance
- Attributes can be set to anything

```
"file t31.py"

from geom1 import *

i1 = Circle(0, 2, 4)
print 'i1:', i1.radius, i1.area()
i1.radius = -2
print 'i1:', i1.radius, i1.area()
i1.radius = 'garbage'
print 'i1:', i1.radius, i1.area()
```

```
i1: 4 50.2654824574
i1: -2 12.5663706144
i1: garbage
Traceback (most recent call last):
  File "M:/My Documents/Python course/tests/t31.py", line 10, in -toplevel-
    print 'i1:', i1.radius, i1.area()
  File "M:/My Documents/Python course/tests\geom1.py", line 15, in area
    return math.pi * self.radius**2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Changing an instance: attribute add/delete

- An attribute can be added!
- And deleted!

```
"file t32.py"

from geom1 import *

i1 = Circle(0, 2)

i1.colour = 'red'                              # add an instance attribute
print 'i1:', i1.radius, i1.colour

del i1.radius                                  # delete an instance attribute
print 'has i1 radius?', hasattr(i1, 'radius')  # built-in function
print 'i1:', i1.area()
```

```
i1: 1 red
has i1 radius? False
i1:
Traceback (most recent call last):
  File "M:/My Documents/Python course/tests/t32.py", line 11, in -toplevel-
    print 'i1:', i1.area()
  File "M:/My Documents/Python course/tests\geom1.py", line 15, in area
    return math.pi * self.radius**2
AttributeError: Circle instance has no attribute 'radius'
```

Inspecting objects: dir

- Use the built-in function 'dir' to inspect objects
- '__doc__': class documentation string
- '__class__': class for the instance

```
>>> import geom1
>>> i1 = geom1.Circle(0, 0, 2)
>>> dir(i1)
['__doc__', '__init__', '__module__', 'area', 'radius', 'x', 'y']
>>> print i1.__doc__
A 2D circle.
>>> print i1.__class__
geom1.Circle
```

```
>>> type(i1)
<type 'instance'>
>>> type(Circle)
<type 'classobj'>
>>> type(i1.radius)
<type 'int'>
>>> type(i1.area)
<type 'instancemethod'>
```

Equality between objects

- Two kinds of equality:
  - Are the two objects similar in value?
  - Are the two references actually pointing to the same object?

```
>>> a = [1, 2]
>>> b = [1, 2]
>>> a == b              # test whether values are equal
True
>>> a is b              # test whether objects are identical
False
>>> a.append(3)
>>> a == b              # test values again
False
```

Special methods in classes

- Special methods '__xxx__' in classes
- Define custom-made behaviour
- See page 327 in 'Learning Python'

```
"file geom2.py: Module with classes for geometrical shapes, 2nd try"

import math

class Circle:
    # ...some code removed here, for clarity...

    def __repr__(self):                 # better string representation
        return "Circle(%g, %g, radius=%g)" % (self.x, self.y, self.radius)

    def __nonzero__(self):              # is a true circle? else point
        return self.radius != 0

    def __cmp__(self, other):           # compare with other: larger or not?
        return cmp(self.radius, other.radius)
```

Using the special methods, part 1

- Special method definitions are detected by Python
- Built-in functions use them; see documentation

```
"file t33.py"

from geom2 import *

i1 = Circle(0, 2.5)
i2 = Circle(3, 4.02, 0)

print str(i1)
print 'is i1 a circle?:', bool(i1)
print 'is i2 a circle?:', bool(i2)
print 'i1 larger than i2?', i1 > i2       # uses __cmp__, if defined
```

```
Circle(0, 2.5, radius=1)
is i1 a circle?: True
is i2 a circle?: False
i1 larger than i2? True
```

Using the special methods, part 2

- Defining special methods may clarify code tremendously
- But: Stay reasonable 'natural'!

```
"file t34.py"

from geom2 import *

circles = [Circle(1, 3),
           Circle(0, 0, 0.2),
           Circle(-1, -1, 10.0)]
print circles                  # this uses 'repr', which calls __repr__

circles.sort()                 # this uses 'cmp', which calls __cmp__
print circles
```

```
[Circle(1, 3, radius=1), Circle(0, 0, radius=0.2), Circle(-1, -1, radius=10)]
[Circle(0, 0, radius=0.2), Circle(1, 3, radius=1), Circle(-1, -1, radius=10)]
```

Inheritance: Class hierarchies

- Let's define a general 'Shape' class
- 'Circle' is a special case of 'Shape'
- 'Blob' is also a special case of 'Shape'
- Notice: redefinition of 'is_round' in 'Blob'

```
"file geom3.py: Module with classes for geometrical shapes, 2nd try"

import math

class Shape:                            # This is a base class
    "General geometrical shape."

    def is_round(self):
        return True


class Circle(Shape):                    # Circle inherits Shape
    # ...same code as in geom2.py...


class Blob(Shape):
    "An undefined blob."

    def is_round(self):                 # overrides method from Shape
        return False
```

Instances of classes using inheritance

- Which method is called by which instance?
- Polymorphism
- Selection of method depends on actual class of the instance
- Extremely powerful, if properly designed class hierarchy

```
"file t35.py"

from geom3 import *

shapes = [Shape(),
          Circle(1, -2),
          Blob()]

for s in shapes:
    print s, 'round?', s.is_round()
```

```
<geom3.Shape instance at 0x009CEF80> round? True
Circle(1, -2, radius=1) round? True
<geom3.Blob instance at 0x009CEF58> round? False
```

Example: Handle sequences, part 1

```
"file bioseq.py  Module to handle NT or AA sequences. Incomplete."

class Bioseq:

    def __init__(self, seq=None):
        self.seq = seq

    def fetch(self, acc):
        pass                # to be defined in inheriting classes


class Nucleotide(Bioseq):

    def fetch(self, acc):
        pass                # code to fetch from EMBL; cause IOError if not

    def translate(self):
        pass                # code to translate NT seq to AA
        return Protein(seq='whatever')


class Protein(Bioseq):

    def fetch(self, acc):
        pass                # code to fetch from Swiss-Prot; cause IOError
```

Example: Handle sequences, part 2

- Write a help function for fetching either 'Nucleotide' or 'Protein'
- This is a so-called factory function

```
"file t36.py"

from bioseq import *

def fetch_seq(acc):
    for cls in [Nucleotide, Protein]:
        try:
            result = cls()
            result.fetch(acc)
            return result
        except IOError:
            pass
    return None

print fetch_seq('A123')
```

```
<bioseq.Nucleotide instance at 0x009CEFA8>
```

Part 6:
Standard library modules

Module 're', part 1

Regular expressions: advanced string patterns

- Define a pattern
  - The pattern syntax is very much like Perl or grep
- Apply it to a string
- Process the results

```
"file t37.py"

import re

seq = "MAKEVFSKRTCACVFHKVHAQPNVGITR"

zinc_finger = re.compile('C.C..H..H')     # compile regular expression pattern
print zinc_finger.findall(seq)

two_charged = re.compile('[DERK][DERK]')
print two_charged.findall(seq)
```

```
['CACVFHKVH']
['KE', 'KR']
```

Module 'sys', part 1

Variables and functions for the Python interpreter

- sys.argv
  - List of command-line arguments; sys.argv[0] is script name
- sys.path
  - List of directory names, where modules are searched for
- sys.platform
  - String to identify type of computer system

```
>>> import sys
>>> sys.platform
'win32'
```

Module 'sys', part 2

- sys.stdout, sys.stdin, sys.stderr
  - Predefined file objects for input/output
  - 'print' stuff goes to 'sys.stdout'
  - May be set to other files
- sys.exit(n)
  - Force exit from Python execution
  - 'n' is an integer error code, normally 0

```
>>> import sys
>>> sys.stdout.write('the hard way')
the hard way
```

Module 'os', part 1

Portable interface to operating-system services

- os.getcwd()
  - Returns the current directory

```
>>> os.getcwd()
'M:\\My Documents\\Python course\\tests'
```

- os.environ
  - Dictionary containing the current environment variables

```
>>> for k, v in os.environ.items(): print k, v

TMP C:\DOCUME~1\se22312\LOCALS~1\Temp
COMPUTERNAME WS101778
USERDOMAIN BIOVITRUM
COMMONPROGRAMFILES C:\Program Files\Common Files
PROCESSOR_IDENTIFIER x86 Family 6 Model 9 Stepping 5, GenuineIntel
PROGRAMFILES C:\Program Files
PROCESSOR_REVISION 0905
HOME C:\emacs
...
```

Module 'os', part 2

- os.chdir(path)
  - Changes the current working directory to 'path'
- os.listdir(path)
  - Return a list of the contents of the directory 'path'

- os.mkdir(path)
  - Create the directory 'path'
- os.rmdir(path)
  - Remove the directory 'path'

- os.remove(path)
  - Remove the file named 'path'

Module 'os', part 3

- os.system(command)
  - Execute the shell command (string) in a subprocess
  - Return the error code as integer
- os.popen(command, mode='r')
  - Run the shell command (string)
  - Open a pipe to the command, return as a file object
  - Mode is either read, or write; not both
- os.popen2, os.popen3, os.popen4
  - Variants of os.popen, with different file objects

- os.getpid()
  - Return the process ID as integer

Module 'os.path', part 1

Portable path name handling

- os.path.abspath(path)
  - Returns the absolute path for the given relative 'path'

```
>>> d = os.path.abspath('.')
>>> d
'M:\\My Documents\\Python course\\tests'
```

- os.path.dirname(path)
  - Returns the directory name part of 'path'

```
>>> os.path.dirname(d)
'M:\\My Documents\\Python course'
```

Module 'os.path', part 2

- os.path.join(path, path, …)
  - Joint together the path parts intelligently into a valid path name

```
>>> d = os.path.join(os.getcwd(), 't1.py')
>>> d
'M:\\My Documents\\Python course\\tests\\t1.py'
```

- os.path.split(path)
  - Splits up the path into directory name and filename
  - Reverse of 'os.path.join'

```
>>> os.path.split(d)
('M:\\My Documents\\Python course\\tests', 't1.py')
```

- os.path.splitext(path)
  - Splits up the path into base filename and the extension (if any)

```
>>> >>> os.path.splitext(d)
('M:\\My Documents\\Python course\\tests\\t1', '.py')
```

Module 'os.path', part 3

- os.path.exists(path)
  - Does the 'path' exist? File, or directory

```
>>> d = os.path.join(os.getcwd(), 't1.py')
>>> os.path.exists(d)
True
```

- os.path.isfile(path)
  - Is 'path' the name of a file?
- os.path.isdir(path)
  - Is 'path' the name of a directory?

```
>>> os.path.isfile(d)
True
>>> os.path.isdir(d)
False
```

- os.path.walk(path, func, arg)
  - Used to process an entire directory tree
  - Visits each subdirectory
  - Calls the function 'func' with a list of the filenames in that directory

Some other useful modules

shutil	'shell utilities': copy files, remove entire directories
StringIO cStringIO	String-like objects that behave as files
gzip, zipfile	Handle compressed files
dbm, gdbm	Files that behave as dictionaries; simple databases
time	Time handling and formatting
random	Random number generator
urlparse urllib, urllib2 ftplib	URL address manipulation URL data fetching, using file-like objects FTP handling
cgi	CGI script operations
Tkinter	Interface to Tk/Tcl; graphical user interface (GUI) module
xml	Package for handling XML files