Python Crash Course

João Pedro Pedroso

Departmento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
jpp@fc.up.pt

Tokyo University of Marine Science and Technology
December 2008

Programming languages

- Machine languages:

```
+15829387589
+18490298492
+17204890938
```

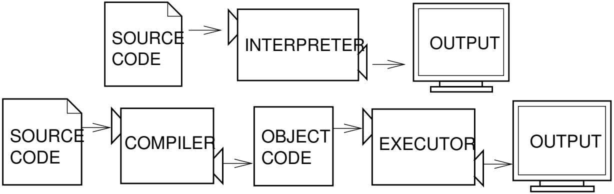
- Assembly: elementary operations represented by abbreviations

```
LOAD X
ADD Y
STORE SOMA
```

- High level languages:

```
Sum = X + Y
```

Compilers and interpreters



Python language

- High level programming language
- Interpreted
- May be used in command-line mode or in script mode
- Has support for object oriented programming
- Convention: program filenames have the extension `.py`

A computer program

input: Get data from the keyboard, a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

The first program

```
print "Hello, World!"
```

Values

- **values:** fundamental things manipulated by the programs

```
>>> print 4
4
>>> type("Hello, World!")
<type 'string'>
>>> type(17)
<type 'int'>
```

Variables

- **variables:** names that refer to values

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
>>> print message
What's up, Doc?
>>> print n
17
>>> print pi
3.14159
```

- Variable names must begin with a letter, and may have an arbitrary number of letters and numbers
- The underscore sign _ is treated as a letter

Statements

- **Statements:** instructions that the Python interpreter can execute
- Can be typed on the command line or in scripts
- When there are several statements, the results appear one at a time, as the statements execute

  ```
  print 1
  x = 2
  print x
  ```

  output:

  ```
  1
  2
  ```

  (the assignment statement produces no output)

Evaluating expressions I

- if we type an expression on the command line, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

- values and variables are considered as expressions

```
>>> 17
17
>>> x
2
```

- evaluating an expression is not the same thing as printing a value:

```
>>> message = "What's up, Doc?"
>>> message
"What's up, Doc?"
>>> print message
What's up, Doc?
```

- When Python displays the value of an expression, it uses the same format you would use to enter a value.

Evaluating expressions II

- In the case of strings, that means that it includes the quotation marks.
- But the print statement prints the value of the expression, which in this case is the contents of the string.
- In a script, an expression all by itself is a legal statement, but it doesn't do anything. The following script produces no output at all:

```
17
3.2
"Hello, World!"
1 + 1
```

- How can we change the script to display the values of these four expressions?

Arithmetic operations

- **Operators** are special symbols that represent computations like addition and multiplication.
- The values the operator uses are called **operands**.

  `20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)`

- +, −, and /, parenthesis: same meaning as in mathematics
- multiplication: `*`
- exponentiation: `**`
- variable names: replaced with their value before the operation is performed.
- Division:

```
from __future__ import division
minute = 59
print minute/60 # floating point division
print minute//60 # integer division
```

Operators and operands

Operation order:

- **P** parenthesis
- **E** exponentiation
- **MD** multiplication, division
- **AS** addition, subtraction

Operators with the same precedence: evaluation from *left* to *right*

Operations on strings

- + ↔ concatenation

```
fruit = "banana"
bakedGood = " nut bread"
print fruit + bakedGood
```

output:

```
banana nut bread
```

- * ↔ repetition

```
fruit = "banana"
print 2 * fruit
```

output:

```
bananabanana
```

More...

- Composition: combining expressions and statements

```
print "Number of minutes since midnight: ", hour*60+minute
```

- Comments:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

Can also be put at the end of the line:

```
percentage = (minute * 100) // 60     # caution: integer division
```

Functions

- we have already seen an example of a **function call**:

```
>>> type("32")
<type 'string'>
```

- name of this function: `type`
- another example: `id`, returns a unique identifier for a value

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

`id` of a variable: `id` of the value to which it refers

Type conversion

- there are built-in functions that convert values from one type to another:

```
>>> int("32")
32
>>> int(3.99999)
3
>>> int(-2.3)
-2
>>> int("Hello")
ValueError: invalid literal for int(): Hello
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Math functions

- defined in the module `math`

```
>>> import math
>>> math.sqrt(2) / 2.0
0.707106781187
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2
```

- composition:

```
>>> x = math.cos(angle + math.pi/2)
```

Adding new functions

```
def NAME( LIST OF PARAMETERS ):
  STATEMENTS
```

Example:

```
def newLine():
  print

print "First Line."
newLine()
newLine()
newLine()
print "Second Line."
```

output:

```
First line.



Second line.
```

Parameters and arguments

```
def printTwice(bruce):
  print bruce, bruce
```

usage:

```
>>> printTwice('Spam')
Spam Spam
>>> printTwice(5)
5 5
>>> printTwice(3.14159)
3.14159 3.14159
>>> printTwice('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> printTwice(math.cos(math.pi))
-1.0 -1.0
```

Variables and parameters are local

- a variable created inside a function only exists inside the function

```
def printTwice(bruce):
  print bruce, bruce
```

```
>>> printTwice('Spam')
Spam Spam
>>> print bruce
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'bruce' is not defined
>>>
```

Conditionals and recursion

- The modulus operator

```
>>> quotient = 7 // 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
>>>
```

- to check whether one number is divisible by another:
  if `x % y` is zero, then `x` is divisible by `y`.
- Boolean expressions

```
>>> 5 == 5
True
>>> 5 == 6
False
```

- comparison operators:

```
      x != y          # x is not equal to y
      x > y           # x is greater than y
      x < y           # x is less than y
      x >= y          # x is greater than or equal to y
```

Conditional execution

```
if x > 0:
  print "x is positive"
```

```
HEADER:
  FIRST STATEMENT
  ...
  LAST STATEMENT
```

Alternative execution

```
if x%2 == 0:
  print x, "is even"
else:
  print x, "is odd"
```

chained conditionals

```
if choice == 'A':
  functionA()
elif choice == 'B':
  functionB()
elif choice == 'C':
  functionC()
else:
  print "Invalid choice."
```

Nested conditionals

```
if x == y:
  print x, "and", y, "are equal"
else:
  if x < y:
    print x, "is less than", y
  else:
    print x, "is greater than", y
```

The return statement

```
import math

def printLogarithm(x):
  if x <= 0:
    print "Positive numbers only, please."
    return

  result = math.log(x)
  print "The log of x is", result
```

Recursion

```
def countdown(n):
  if n == 0:
    print "Blastoff!"
  else:
    print n
    countdown(n-1)
```

example:

```
>>> countdown(3)
3
2
1
Blastoff!
```

Keyboard input

```
>>> input = raw_input ()
What are you waiting for?
>>> print input
What are you waiting for?
```

Fruitful functions

- **return values:** calling a function may produce a result

```
e = math.exp(1.0)
height = radius * math.sin(angle)
```

- a function produces a result with `return`

```
import math

def area(radius):
  temp = math.pi * radius**2
  return temp
```

- another example:

```
def absoluteValue(x):
  if x < 0:
    return -x
  else:
    return x
```

Boolean functions

- return `True` or `False` (or `1` or `0`)

```
def isDivisible(x, y):
  return x % y == 0
```

More recursion:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

Fibonacci function

```
def fibonacci (n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Checking types

- what happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial (1.5)
RuntimeError: Maximum recursion depth exceeded
```

- values of `n` *miss* the base case (`n == 0`)
- corrected version:

```
def factorial (n):
  if type(n) != type(1):
    print "Factorial is only defined for integers."
    return -1
  elif n < 0:
    print "Factorial is only defined for positive integers."
    return -1
  elif n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Functions

what are functions good for?

- Giving a name to a sequence of statements makes a program easier to read and debug.
- Dividing a long program into functions allows to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Well-designed functions are often useful for many programs. Once we write and debug one, we can reuse it.
- Functions facilitate both recursion and iteration.

Iteration: the while statement

- recursive version of `countdown`:

```
def countdown(n):
  if n == 0:
    print "Blastoff!"
  else:
    print n
    countdown(n-1)
```

- iterative version with `while`:

```
def countdown(n):
  while n > 0:
    print n
    n = n-1
  print "Blastoff!"
```

Iteration: the while statement

- flow of execution for a `while` statement:
  1. Evaluate the condition, yielding `False` or `True`.
  2. If the condition is `False` (0), exit the `while` statement and continue execution at the next statement.
  3. If the condition is `True` (1), execute each of the statements in the body and then go back to step 1.

  (body: all the statements below the header with the same indentation)
- Note: the body must change the value of some variable, so that the condition becomes false and the loop terminates
- Otherwise: **infinite loop**

Iteration: the while statement

```
def sequence(n):
  while n != 1:
    print n,
    if n%2 == 0:          # n is even
      n = n/2
    else:                 # n is odd
      n = n*3+1
```

When does this function terminate?

Strings

- *Strings* are different of `int`s and `float`s, because they are made of smaller pieces (*characters*)
- bracket operator: selects a single character from a string

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

  what is the output?
- the *zero-th* character of `"banana"` is `b`
- `len` function: returns the number of characters in a string

```
>>> fruit = "banana"
>>> len(fruit)
6
```

- accessing the last element of a string:

```
length = len(fruit)
last = fruit[length]        # ERROR!
length = len(fruit)
last = fruit[length-1]
```

- alternatively: use negative indices

```
fruit[-1]          # yields the last letter
fruit[-2]          # yields the second last letter
```

The for loop

- One way of traversing a string

```
index = 0
while index < len(fruit):
  letter = fruit[index]
  print letter
  index = index + 1
```

- a simpler syntax: the `for` loop

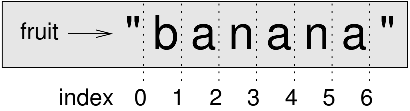
```
for char in fruit:
  print char
```

String slices

- a segment of a string is called a *slice*

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

- `banana` string



- omitting the first or the last indices:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

String comparison

- comparison operators work on strings

```
if word == "banana":
  print  "Yes, we have no bananas!"
```

- putting words in alphabetical order:

```
if word < "banana":
  print "Your word," + word + ", comes before banana."
elif word > "banana":
  print "Your word," + word + ", comes after banana."
else:
  print "Yes, we have no bananas!"
```

- problem: in python uppercase letters come before lowercase letters

```
Your word, Zebra, comes before banana.
```

Strings are immutable

```
greeting = "Hello, world!"
greeting[0] = 'J'            # ERROR!
print greeting
```

we can't change an existing string

(one solution:)

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

Functions with strings

- the `find` function

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

- counting

```
fruit = "banana"
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

Built-in functions on strings I

- finding characters:

```
>>> fruit = "banana"
>>> index = fruit.find("a")
>>> print index
1
>>> index = fruit.find("na")
>>> print index
2
>>> index = fruit.find("na", 3)
>>> print index
4
```

- replacing:

```
>>> r = fruit.replace("na", "pa")
>>> print r
bapapa
```

- changing and checking lower/upper case:

Built-in functions on strings II

```
>>> b = fruit.upper()
>>> b
'BANANA'
>>> b.isupper()
True
>>> fruit.isupper()
False
```

Lists

- a **list** is an ordered set of values, where each value is identified by an index
- are similar to strings, but list's elements can have any type
- examples:

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
["hello", 2.0, 5, [10, 20]]
```

- *nested list:* A list within another list
- empty list: `[]`
- assignment to variables:

```
vocabulary = ["ameliorate", "castigate", "defenestrate"]
numbers = [17, 123]
empty = []
print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
```

range

- lists that contain consecutive integers

```
>>> range(1,5)
[1, 2, 3, 4]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Accessing elements

- syntax: the same as for accessing characters in strings

```
>>> numbers = [17, 123]
>>> numbers[0]
17
>>> numbers[-1]
123
>>> numbers[:]
[17, 123]
```

- read or write an element that does not exist: runtime error:

```
>>> numbers[2] = 5
IndexError: list assignment index out of range
>>> numbers[-2]
17
>>> numbers[-3]
IndexError: list index out of range
```

List membership

- `in` is a boolean operator that tests membership in a sequence

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
True
>>> 'debauchery' in horsemen
False
>>> 'debauchery' not in horsemen
True
```

Iteration

- with `while`

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < len(horsemen):
  print horsemen[i]
  i = i + 1
```

- with `for`

```
horsemen = ["war", "famine", "pestilence", "death"]

for horseman in horsemen:
  print horseman
```

Iteration

- with `for`

```
for VARIABLE in LIST:
  BODY
```

- with `while`

```
i = 0
while i < len(LIST):
  VARIABLE = LIST[i]
  BODY
  i = i + 1
```

Operations on lists

- + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

- * operator repeats a list:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List slices

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e', 'f']
>>> list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Lists are mutable

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']

...

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = ['x', 'y']
>>> print list
['a', 'x', 'y', 'd', 'e', 'f']

...

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = []
>>> print list
['a', 'd', 'e', 'f']
```

List deletion

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del list[0]
>>> list
['b', 'c', 'd', 'e', 'f']
>>> del list[1:3]
>>> list
['b', 'e', 'f']
```

Built-in functions on lists I

- appending:

```
>>> r = [1,2,3]
>>> r.append(1)
>>> r
[1, 2, 3, 1]
```

- sorting

```
>>> r = [3,1,4,2,6,5]
>>> r.sort()
>>> r
[1, 2, 3, 4, 5, 6]
```

- counting elements:

```
>>> a = [1,2,3,1,2,2]
>>> a.count(2)
3
```

- removing elements:

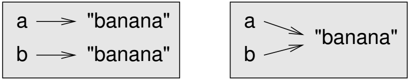
Built-in functions on lists II

```
>>> r = [1,2,3,1,2,3]
>>> r.remove(2)
>>> r
[1, 3, 1, 2, 3]
>>> r.remove(2)
>>> r
[1, 3, 1, 3]
```

Objects and values I

- the following assignment might lead to two possible states

```
a = "banana"
b = "banana"
```



- we can check this with `id`:

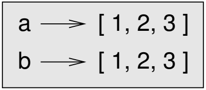
```
>>> id(a)
135044008
>>> id(b)
135044008
```

(`a` and *b* refer to the same string)

- lists behave differently:

Objects and values II

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

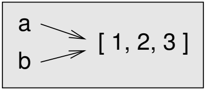


`a` and `b` have the same value but do not refer to the same object

Aliasing

- if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
```



- as the same list has two different names, `a` and `b`, we say that it is **aliased**
- changes made with one alias affect the other:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Cloning lists

- if we want to modify a list and also keep a copy of the original, we need to be able to make a copy (*cloning*)

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

- another possibility for cloning:

```
>>> b = list(a)
```

List parameters

- passing a list as an argument actually passes a reference to the list, not a copy of the list

```
def deleteHead(list):
  del list[0]

...

>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

Matrices

- nested lists are often used to represent matrices

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
```

Strings and lists

there are some useful functions for dealing with lists and strings

- splitting

```
>>> a = "tokyo osaka kyoto"
>>> a.split()
['tokyo', 'osaka', 'kyoto']
>>> a.split('k')
['to', 'yo osa', 'a ', 'yoto']
```

- splitting

```
>>> wines = ['port', 'champagne', 'bordeaux']
>>> " ".join(wines)
'port champagne bordeaux'
>>> " + ".join(wines)
'port + champagne + bordeaux'
```

formating strings

formating strings: the % operator (again!)

```
>>> for i in range(5):
...   print "format: %3d %10f %10s" % (i, 1/(i+1), str(i*100))
...
format:   0   1.000000          0
format:   1   0.500000        100
format:   2   0.333333        200
format:   3   0.250000        300
format:   4   0.200000        400
```

Mutability and tuples

- a **tuple** that is similar to a list, but it is immutable
- syntax: comma-separated list of values
```
>>> t = 'a', 'b', 'c', 'd', 'e'
>>> t = ('a', 'b', 'c', 'd', 'e')     # equivalent representation
```
- for creating a tuple with a single element:
```
>>> t1 = ('a',)     # a tuple
>>> type(t1)
<type 'tuple'>
>>> t2 = ('a')      # a string in parenthesis
>>> type(t2)
<type 'string'>
```
- accessing elements: as in lists:
```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
>>> t[1:3]
('b', 'c')
```
- if we try to modify an element, we get an error:
```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

tuple assignment

- to swap the values of two variables

```
>>> temp = a
>>> a = b
>>> b = temp
```

- with tuple assignment:

```
>>> a, b = b, a
```

- the number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

Tuples as return values

- functions can return tuples as return values

```
def f(x):
  sum = 0
  max = x[0]
  for i in x:
    sum += i
    if i > max:
      max = i
  return sum, max
```

Dictionaries

- strings, lists, and tuples use integers as indices
- **dictionaries** are similar, but they can use any immutable type as an index
- empty dictionary: represented by `{}`

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

- another way:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Dictionary operations

- deletion

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

- change:

```
>>> inventory['apples'] = 0
>>> print inventory
{'oranges': 525, 'apples': 0, 'bananas': 312}
```

- `len` function returns the number of key-value pairs:

```
>>> len(inventory)
3
```

Built-in functions on dictionaries I

- `keys` returns a list of the keys of the dictionary

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

- `values` returns a list of the values in the dictionary:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

- `items` returns a list of tuples, one for each key-value pair:

```
>>> eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

- `has_key` takes a key and returns `True` if the key appears in the dictionary:

```
>>> eng2sp.has_key('one')
True
>>> eng2sp.has_key('deux')
False
```

copying dictionaries

- as dictionaries are mutable (as for lists), we need to be aware of aliasing
- whenever two variables refer to the same object, changes to one affect the other

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

- another possibility for copying:

```
>>> copy = dict(opposites)
```

Sparse matrices

- sparse matrices have most of their elements equal to zero
- lists of lists might not be the appropriate way to represent them

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]
```

- using dictionaries is more economical:

```
>>> matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
>>> matrix[0,3]
1
```

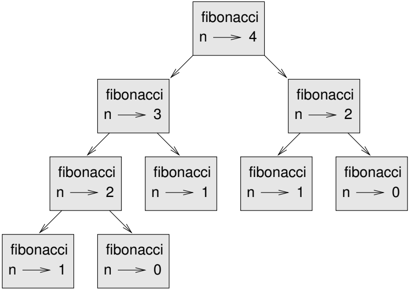
- accessing elements which are not stored in the dictionary gives an error:

```
>>> matrix[1,3]
KeyError: (1, 3)
```

- solution: `get` method

```
>>> matrix.get((0,3), 0)
1
>>> matrix.get((1,3), 0)
0
```

Fibonacci again I



- `fibonacci(0)` and `fibonacci(1)` are called many times!
- solution: is to keep track of values that have already been computed by storing them in a dictionary

Fibonacci again II

```
previous = {0:1, 1:1}

def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

- computation is now much quicker:

```
>>> fibonacci(50)
20365011074L
>>> fibonacci(500)
225591516161936330872512695036072072046011324913758190588638866418
```

Random numbers

- in some applications (like games) we want the computer to be unpredictable
- we can generate random numbers and use them to determine the outcome of the program
- Python provides a pseudo-random generator in the module `random`

```
import random
def randomList(n):
  s = [0] * n
  for i in range(n):
    s[i] = random.random()
  return s
>>> randomList(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

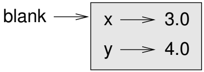
Object oriented programming

- we can define new types, using the `class` keyword:

```
class Point:
  pass

...

>>> blank = Point()
>>> blank.x = 3.0
>>> blank.y = 4.0
```




```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

Example: a class for representing time

```
class Time:
  pass

def printTime(time):
  print str(time.hours) + ":" +
        str(time.minutes) + ":" +
        str(time.seconds)

...
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
```

Methods: functions defined in a class

```
class Time:
  def printTime(time):
    print str(time.hours) + ":" +
          str(time.minutes) + ":" +
          str(time.seconds)
...
>>> currentTime.printTime()
```

Special methods

- initialization:

```
class Time:
  def __init__(self, hours=0, minutes=0, seconds=0):
    self.hours = hours
    self.minutes = minutes
    self.seconds = seconds
```

- the `__init__` method is called whenever we create a `Time` object:

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
>>> 9:14:30
```

- because the parameters are optional, we can omit them:

```
>>> currentTime = Time()
>>> currentTime.printTime()
>>> 0:0:0
```

More attributes on the point class

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'

...
>>> p = Point(3, 4)
>>> print p
(3, 4)
```

Object oriented features

- Some times, the programs are more clean without object-orientations
- Other times, object orientation simplifies a lot the operations
- We must decide case-by-case which is more appropriate

For Further Reading

- Allen B. Downey, Jeffrey Elkner, and Chris Meyers.
  How to think like a computer scientist.
  http://www.thinkpython.com
- Guido van Rossum et al.
  Python documentation
  http://www.python.org