Python

Carl Kingsford - 648b

Basic Structure

Python is an interpreted language (like Perl).

Programs are in files with the .py extension.

Programs should start with a “#!” line:

```
#!/usr/bin/env python
```

Programs are executed from top to bottom.

Advanced: it’s strongly dynamically typed (values have a fixed type, but variables can change type on the fly.)

Most unusual syntax: indenting and newlines are important.

Unlike Perl, there are no { } characters to indicate the start and end of a block. That is done through indenting.

Interactive Mode

The command “python” will start an interactive python session:

```
$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can enter any python commands here.

The most important one is help(x), which will show you detailed help on function (or type or class) x.

Use Ctrl-D or quit() to exit.

Example

```
#!/usr/bin/env python

import sys
import seq

def remove_gap(s):
    return s.replace('-','')

S1 = seq.read_fasta(sys.argv[1])
S2 = seq.read_fasta(sys.argv[2])


print sys.argv[1]
print sys.argv[2]


SD1 = dict((s.name, s) for s in S1)
SD2 = dict((s.name, s) for s in S2)

assert len(SD1) == len(SD2)

for s in SD1.itervalues():
    if s.seq != SD2[s.name].seq:
        print 'DISAGREE:', s.name
        print s.seq
        print SD2[s.name].seq
    if s.seq == SD2[s.name].seq:
        print 'AGREE:', s.name
```

Import some libraries (sys is a standard one; seq is one I wrote)

Define a function

Call the function “read_fasta” in the seq library.

Print some info to the screen

Create some dictionary data structures (called hashes in Perl) that map sequence names to DNA sequences.

For every sequence in the dictionary SD1, check that the corresponding sequence in SD2 matches

Example 2

A function that takes 1 parameter

“Docstring” that documents what the function does.

```
def random_order(n):
    “Create random mapping between [n] and [n]”
    import random
    R = range(n)
    random.shuffle(R)
    return dict(enumerate(R))
```

Load the “random” library.

R = [0, 1, 2, 3, ..., n-1]

The list R is randomly shuffled to be something like [7, 8, 10, n-1, ..., 4]

Turns list of pairs [(i,j)] into a mapping from i → j

Turns shuffled list into a list of pairs:
[(0, 7), (1, 8), (2, 10), ...]

Data Structures

Main Idea: Sequences

Built-in Basic Data Types

str = string (delimit with ‘xyz’ or “xyz”)

```
>>> str(10)
'10'
```

int = arbitrary-sized integer (see also long)

```
>>> 7**73
49221735352184872959961855190338177606846542622561400857
262407L
```

float = floating point number

```
>>> 1/2
0
>>> 1.0/2
0.5
```

bool = True or False

```
>>> bool(10)
True
>>> bool(0)
False
```

Collection Data Types

list = mutable list

```
>>> ['a','b',10,10,7]
['a', 'b', 10, 10, 7]
```

tuple = frozen list (can’t change)

```
>>> ('a','b',10, 10,7)
('a', 'b', 10, 10, 7)
```

dict = dictionary, aka hash

```
>>> {'a':7, 'b':10, 13:2}
{'a': 7, 'b': 10, 13: 2}
```

set = mutable set of elements

```
>>> set(['a','b','b',10])
set(['a', 10, 'b'])
```

frozenset = frozen set of elements

```
>>> frozenset(['a','b','b',10])
frozenset(['a', 10, 'b'])
```

Collections

Can contain items of different type.

Can nest them: [(1, 2), (3, 4), [5, 6, 7, 8], {'a': 2}]

Sets do not preserve order.

Dictionary keys must be constant, but can be frozenset or tuples:

```
>>> A = {}
>>> A[(1,2)] = 10
>>> A[frozenset([2,2,2,2])] = 13
>>> A
{(1, 2): 10, frozenset([2]): 13}
>>> A[ [10,2] ] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Slicing Lists and Strings

Can extract subranges from lists and strings:

```
s = “abcdef”
s[0]  == “a”
s[2:4] == “cd”
s[2:] == “cdef”
s[-1] == “f”
```

negative numbers count from the end.

```
L = [1,2,3,4,5]
L[3:7] == [4,5]
L[:2] == [1,2]
T = (7,8,9,10)
T[1:3] == (8,9)
```

Note: range i:j gives characters i, i+1,..., j-1.

For range i:j

if i is omitted, it’s assumed to be 0.
if j is omitted, it’s assumed to be len + 1.

Assignment works for lists (but not strings or tuples):

```
L[2:4] =  [7,8,9,10] → [1, 2, 7, 8, 9, 10, 5]
```

For Loops

For loops always loop over a sequence.

Collections are sequences.

```
for x in [1,2,3,4]:
        print x
```

Prints 1 2 3 4

```
for key in {‘a’:10,’b’:100}:
        print key
```

Prints a b OR   b a

```
for i in set([1,2,3,2]):
        print i
```

Prints 1 2 3 in some order

Generate sequences:

```
range(100) = [0,1,2,...,99]
range(10,50) = [10,11,...,49]
range(10,20,2) = [10, 12, 14, 16, 18]
```

```
for i in range(32):
     print 2**i
```

Local Alignment Python Code

```
def local_align(x, y, score=ScoreParam(-7, 10, -5)):
    """Do a local alignment between x and y"""
    # create a zero-filled matrix
    A = make_matrix(len(x) + 1, len(y) + 1)

    best = 0
    optloc = (0,0)

    # fill in A in the right order
    for i in xrange(1, len(x)):
        for j in xrange(1, len(y)):

            # the local alignment recurrance rule:
            A[i][j] = max(
               A[i][j-1] + score.gap,
               A[i-1][j] + score.gap,
               A[i-1][j-1] + (score.match if x[i] == y[j] else score.mismatch),
               0
            )

            # track the cell with the largest score
            if A[i][j] >= best:
                best = A[i][j]
                optloc = (i,j)

    # return the opt score and the best location
    return best, optloc
```

List Comprehensions

Can construct lists from rules:

```
L = [i**2 + j**2 for i in range(10)
        for j in range(10)
             if i >= j]
```

```
>>> L
[1, 4, 5, 9, 10, 13, 16, 17, 20, 25, 25, 26, 29, 34, 41, 36, 37, 40, 45, 52, 61, 49, 50, 53, 58, 65,
74, 85, 64, 65, 68, 73, 80, 89, 100, 113, 81, 82, 85, 90, 97, 106, 117, 130, 145]
>>> set(L)
set([1, 130, 4, 5, 9, 10, 13, 16, 17, 20, 25, 26, 29, 34, 36, 37, 40, 41, 45, 49, 50, 52, 53, 58,
61, 64, 65, 68, 73, 74, 80, 81, 82, 85, 89, 90, 97, 100, 145, 106, 113, 117])
```

General syntax: [EXPR for … if … for … if]

```
L = []
for i in range(10):
        for j in range(10):
             if i >= j:
                  L.append(i**2 + j**2)
```

Local Alignment Python Code

```
def make_matrix(sizex, sizey):
    """Creates a sizex by sizey matrix filled with zeros."""
    return [[0]*sizey for i in xrange(sizex)]


class ScoreParam:
    """The parameters for an alignment scoring function"""
    def __init__(self, gap, match, mismatch):
        self.gap = gap
        self.match = match
        self.mismatch = mismatch
```

Generators

Often it is wasteful to create a list in memory:

```
for i in range(2**20):
        print i
```

First creates a list of ≈ 1 million items, then iterates through it.

```
for i in xrange(2**20):
        print i
```

Creates a generator for the list and iterates through it.

Generators are rules that generate a sequence:

```
(i**2 + j**2 for i in range(10)
        for j in range(10)
             if i >= j)
```

Generator has same syntax as list comprehension, but will only create an item as you iterate through it.

The only thing you can do with generators is iterate through them.

Composing Generators

Generators and other sequences can be passed to functions that create new generators:

```
G = (i**2 + j**2 for i in xrange(10) for j in xrange(10) if i >= j)
for i in sorted(G):
        print i
```

G is a saved generator
sorted(G) returns the same
sequence as G, but sorted

```
s = “abcd”
for c in reversed(s):
        print c
```

s → (‘d’, ‘c’, ‘b’, ‘a’)

```
L = [“a”, “b”, “c”, “d”]
for (i, c) in enumerate(L):
        print i, c
```

L → ((0, “a”), (1, “b”), (2, “c”), (3, “d”))

```
Q = [“e”, “f”, “g”, “h”]
for (a,b) in zip(Q, L):
        print a,b
```

((“e”, “a”), (“f”, “b”), (“g”, “c”), (“h”, “d”))

Organizing Code

Functions

Functions can be defined using the syntax:

```
def name(a, b, c=True, d=2*10):
    BODY
```

The syntax “= EXPR” after a parameter gives the parameter’s default value.

Functions can be called using:

```
name(10,20, False)
name(10, b=20, d=32)
name(b=10, a=20)
```

Values can be returned from functions using the return statement:

```
def sum(S):
    s = 0.0
    for i in S: s = s + i
    return s
```

Comments

Comments start with # and go until the end of the line:

```
# this is a comment
```

Strings can be placed as comments as first statement in a file or a function:

```
def bandwidth(M):
    "Compute the Bandwidth of M"
    return max(abs(i-j) for i in xrange(len(M))
        for j in xrange(i,len(M)) if M[i,j] != 0)
```

Strings surrounded by “””xxx””” or ‘’‘xxx’’’ can span multiple lines.

Packages

Code can be imported from other files and standard packages using import:

```
import NAME
from NAME import id1, id2, id3 ...
from NAME import *
```

For example:

```
import math
print math.log(10)
from math import log
print log(10)
```

import will search your current directory, the standard python directories, and directories in your PYTHONPATH environment variable.

Classes

A class represents a user defined type.

Classes can have functions and variables associated with them.

Classes are instantiated into objects.

```
class Species:
        def __init__(self, name):
            self.name = name

        def species_name(self):
            return self.name
```

Special function called _ _init_ _ is the constructor that says how to build an instance of the class.

All functions in a class take a “self” parameter that represents the object.

```
Ce = Species(“C. elegans”)
Hs = Species(“H. sapiens”)
```

New instance of Species created with name = “C. elegans”

```
print Ce.name, Hs.name
print Ce.species_name(), Hs.species_name()
```

Classes

Objects made from classes can be used anywhere other variables can be used:

```
L = [Hs, Ce, Hs]

Strange = Species(Hs)
```

Syntactically correct!

Fields can be added to objects on the fly:

```
Hs.size = 10
print Hs.size
print Ce.size
```

Error! “size” field only exists in the Hs object.

Classes

```
class TreeNode:
    """Represents a node in the tree to be drawn"""

    def __init__(self, parent=None, name="", **options):
        self.name, self.parent = name, parent
        self.children = []
        self.length = 0.0

        if parent != None: parent.children.append(self)
        if "default_len" in options:
            self.length = options["default_len"]
```

Python Code to Build a Suffix Trie

```
class SuffixNode:
    def __init__(self, suffix_link = None):
        self.children = {}
        if suffix_link is not None:
           self.suffix_link = suffix_link
        else:
           self.suffix_link = self

    def add_link(self, c, v):
        """link this node to node v via string c"""
        self.children[c] = v
```

```
def build_suffix_trie(s):
    """Construct a suffix trie."""
    assert len(s) > 0

    # explicitly build the two-node suffix tree
    Root = SuffixNode()      # the root node
    Longest = SuffixNode(suffix_link = Root)
    Root.add_link(s[0], Longest)

    # for every character left in the string
    for c in s[1:]:
        Current = Longest; Previous = None
        while c not in Current.children:

            # create new node r1 with transition Current -c->r1
            r1 = SuffixNode()
            Current.add_link(c, r1)

            # if we came from some previous node, make that
            # node's suffix link point here
            if Previous is not None:
                Previous.suffix_link = r1

            # walk down the suffix links
            Previous = r1
            Current = Current.suffix_link

        # make the last suffix link
        if Current is Root:
            Previous.suffix_link = Root
        else:
            Previous.suffix_link = Current.children[c]

        # move to the newly added child of the longest path
        # (which is the new longest path)
        Longest = Longest.children[c]
    return Root
```

Other Statements

Reading Files

“with” statement sets up a context. The main use is to open an file and ensure, no matter what happens, the file will be closed.

```
with open(filename) as inp:
    for line in inp:
        line = line.strip()
        s = line.split()
        …
```

Input file is a sequence of lines & we can iterate over the lines using a for loop

the strip() function removes whitespace from the start and end of the string

split() converts the string into a list of words

Print

```
print expr1, expr2, ..., exprK
```

will output the result of converting the given expressions into strings.

Expressions will be separated by a space, and a newline will be printed at the end.

```
>>> print 10, 20, “cat”, 2*100-5
10 20 cat 195
```

End with a comma to omit the newline at the end and to smartly separate items with spaces:

```
>>> for a in (1,2,3,4): print “item=”, a,
item= 1 item= 2 item= 3 item= 4
```

Output to a file with the (strange) syntax:

```
print >>F, expr1, expr2, ..., exprK
```

where F is an open file object.

Math Operators

x + y; x - y; x * y: addition, subtraction, and multiplication

x / y	: type-preserving division (if x and y are both integers, the result will be an integer)
x // y	: integer division (floor(float(x)/y))
x % y	: remainder of x / y
x**y	: x raised to the y^{th} power
abs(x)	: absolute value of x
round(x)	: round x to nearest integer
sum(SEQ)	: sum of items in the sequence
max(SEQ)	: largest item in the sequence
min(SEQ)	: smallest item in the sequence

floor, ceil, log, exp, sin, cos, sqrt, factorial, and others available in the built-in “math” package.

Boolean Expressions

Comparison operators are: ==   <   > <=   >=   !=   in   is

```
>>> 1 == 2
False
>>> 1 > 2
False
>>> 1 <= 2
True
>>> 1 != 2
True
>>> "a" in “aeiou”
True
>>> 7 in [7,8,9]
True
```

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a == b
True
>>> a is b
False
>>> 4 not in b
True
>>> i = 10
>>> 0 < i < 100
True
```

Boolean operators are: and or not

```
“a” in “aeiou” and “z” not in “aeiou”

1 < i < 128 and i*j == 100
```

If Statements

```
if 2 in xrange(-3,10,2):
    print “YES”
```

Syntax: if EXPR:

```
if “abc” in “abcde”:
    print “YES”
else:
    print “NO”
```

“else” block executed if the if-EXPR is False.

```
if s == “Whitman”:
    print “Leaves of Grass”
elif s == “Poe”:
    print “The Raven”
elif s == “Hawthorne”
    print “The House of Seven Gables”
else:
    print “Author unknown”
```

“elif” blocks are tested in order if the first if is False and the first elif block that is True is run.

While Loops

```
while EXPR:
    BLOCK
```

will repeatedly execute BLOCK until EXPR is False.

continue: jump to the next iteration of the while or for loop.

break: exit out of the while or for loop.

Regular Expressions

```
import re
S = “al capone abalone”
if re.search(r’one|all$’, S):
        print “FOUND”
```

r‘ ’ strings don’t treat \ as a special character

The results of the search can be saved:

```
m = re.search(r’(.one).*(.one)’, S)
m.group(0) == “pone abalone”
m.group(1) == “pone”
m.group(2) == “lone”
m.start() == 5
m.end() == 17
```

re.sub performs substitutions:

```
S2 = re.sub(r’[aeiou]’, ‘’, S, count=10)
```

Omit count to replace all.
S is unchanged.

re.findall finds all non-overlapping instances:

```
re.findall(r'[aeiou]', S)
['a', 'a', 'o', 'e', 'a', 'a', 'o', 'e']
```

Regular Expressions 2

re.split divides the string at the pattern:

```
>>> re.split(r'[\s,]*', "10 , 200,30 74")
['10', '200', '30', '74']
```

Regular expressions support:

- ^ $: start, end of string
- * : repeat 0 or more times
- + : repeat 1 or more times
- ? : occur 0 or 1 time
- {m,n} : occur between m and n times (inclusive)
- [ ] : character classes
- | : or
- () : grouping for later retrieval
- \number : match contents of given group
- \s : matches space
- \d : matches digit
- \w : matches alphanumeric