Lecture 15
Python: modules, system calls, and containers (revisited)

Course: Practical Bioinformatics (BIOL 4220)
Instructor: Michael Landis
Email: michael.landis@wustl.edu

Lecture 15 outline

Last time: Python strings, files

This time: Python (4 of 4)

Python

- modules
- system calls
- containers (revisited)

Modules

Modules define functions and datatypes that can help solve domain-specific problems

Modules are ***installed*** on a computer then ***imported*** into a Python session to extend the default functionality of the language

```
$ pip install emoji
[ ... installing ... ]

$ python
[ ... initialization text ... ]

>>> import emoji
>>> print(emoji.emojize('Python is :thumbs_up:'))
Python is 👍
```

Anatomy of a module

Modules generally define functions and datatypes, but do not load or process data unless the module is called externally

```
#!/usr/bin/python
import sys

# add two numbers
def add(a, b):
    return a+b

# multiply two numbers
def mult(a, b):
    return a*b

# behavior if called from command line
if __name__ == "__main__":
    import sys
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    z = add(a, b)
    print(f'{z} = {a} + {b}')
```

babymath.py

Using a module

Ways to access module functions and types

```
>>> import babymath              # import module
>>> babymath.add(2,3)
5
```

```
>>> import babymath as bm        # use shortname for module
>>> bm.add(2,3)
5
```

```
>>> from babymath import add     # import one function from module
>>> add(2,3)
5
```

The *__main__()* function will run if the module code is run as a script in Unix

```
$ chmod +x babymath.py
$ ./babymath.py 2 3
5 = 2 + 3
```

Module contents

List module methods using *dir()*;
Print function definitions with *inspect.getsource(f)*

```
>>> import babymath

>>> # list `babymath` module methods
>>> dir(babymath)
['__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__',
 'add', 'mult', 'sys']

>>> # view function definitions
>>> import inspect
>>> print(inspect.getsource(babymath.add))
def add(a, b):
    return a+b

>>> print(inspect.getsource(babymath.mult))
def mult(a, b):
    return a*b
```

Listing object methods

Use *dir()* with any object to list methods its type

```
>>> # methods for list, [1, 2, 3]
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> # methods for string, 'a'
>>> dir('a')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

System calls

Multiple ways to dispatch commands to operating system and retrieve output

```
>>> import os
>>> cmd = 'ls -lart'
>>> out = os.popen(cmd).readlines()
>>> print(''.join(out))
total 12
drwxrwxr-x 10 mlandis mlandis 4096 Nov 10 10:17 ..
-rwxrwxr-x 1 mlandis mlandis 305 Nov 10 12:54 babymath.py
drwxrwxr-x 2 mlandis mlandis 4096 Nov 10 13:38 .
```

using *os.popen()*

```
>>> import subprocess
>>> cmd = 'ls -lart'
>>> p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
>>> out = p.stdout.readlines()
>>> for i,o in enumerate(out):
... out[i] = o.decode('UTF-8')
...
>>> print( ''.join(out) )
total 12
drwxrwxr-x 10 mlandis mlandis 4096 Nov 10 10:17 ..
-rwxrwxr-x 1 mlandis mlandis 305 Nov 10 12:54 babymath.py
drwxrwxr-x 2 mlandis mlandis 4096 Nov 10 13:38 .
```

using *subprocess.Popen()*

Test for element in container

The test "*x in y*" returns True if an element in *y* equals the value of *x*

```
>>> x = [ 1, 2, 3 ]
>>> y = 1
>>> if y in x:
... print(f'{y} is in {x}')
...
1 is in [1, 2, 3]
```

using *in* test with integer list

```
>>> x = 'turducken'
>>> y = 'duck'
>>> if y in x:
... print(f'{y} is in {x}')
...
duck in turducken
```

using *in* test with string

List concatenations

Use the + operator to ***concatenate*** lists with lists, or strings with strings

```
>>> turkey = 'gobble'
>>> duck = 'quack'
>>> turkey + duck + 'bock'
'gobblequackbock'
>>> x = [ 1, 2 ]
>>> y = [ 3, 4 ]
>>> x + y + [ 5, 6 ]
[1, 2, 3, 4, 5, 6]
```

Merge dictionaries

Merge the items in dictionary *y* into dictionary *x*
using the *x.update(y)* method

```
>>> x = { 'cat':'meow', 'dog':'woof' }
>>> y = { 'cow':'moo', 'horse':'neigh' }
>>> x + y
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>> x.update(y)
>>> x
{'cat': 'meow', 'dog': 'woof', 'cow': 'moo', 'horse': 'neigh'}
```

List comprehensions

List comprehensions iterate through each element in a container using a compact notation; returns the processed list

```
>>> # example with list
>>> x = [ 1, 2, 3, 4, 5 ]
>>> # simple list comprehension
>>> y = [ (i**2) for i in x ]
[1, 4, 9, 16, 25]
>>> # list comprehension with if-statement
>>> z = [ (i**2) for i in x if i > 3 ]
[16, 25]
>>> # list comprehension for dictionary
>>> d = {'a':1, 'b':2, 'c':3}
>>> [ f'key={k},val={v}' for k,v in d.items() ]
['key=a,val=1', 'key=b,val=2', 'key=c,val=3']
```

Unpacking lists

Unpack a list to pass *x* as function arguments;
*f(*x)* will treat *x[0]* as arg1, *x[1]* as arg2, *etc.*

```
>>> def add(a,b):
...     return a + b

>>> x = [ 1, 2 ]

>>> # do not unpack `x`
>>> add(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 1 required positional
argument: 'b'

>>> # unpack `x` with `*x`
>>> add(*x)
3
```

Zipped containers

Use *zip(x,y)* to create a ***zipped container***
in which *z[i] = (x[i], y[i])*

```
>>> x = [ 'a', 'b', 'c' ]
>>> y = [ 1, 2, 3 ]
>>> zip(x,y)
<zip object at 0x7f3da6b93880>
>>> list(zip(x,y))
[('a', 1), ('b', 2), ('c', 3)]
>>> for i,j in zip(x,y):
...     print(f'zipped pair {i} and {j}')
zipped pair a and 1
zipped pair b and 2
zipped pair c and 3
```

Index slicing, revisited

The notation *x[i:j:k]* retrieves elements *x[i]* through *x[j-1]* by every *kth* element

```
>>> x = list(range(10))
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> x[3:8:2]
[3, 5, 7]
>>> x[8:3:-2]
[8, 6, 4]
```

slicing list of integers

```
>>> y = 'syzygy'
>>> y[1:3]
'yz'
>>> y[::2]
'szg'
>>> y[::-2]
'yyy'
```

slicing a string

Overview for Lab 15