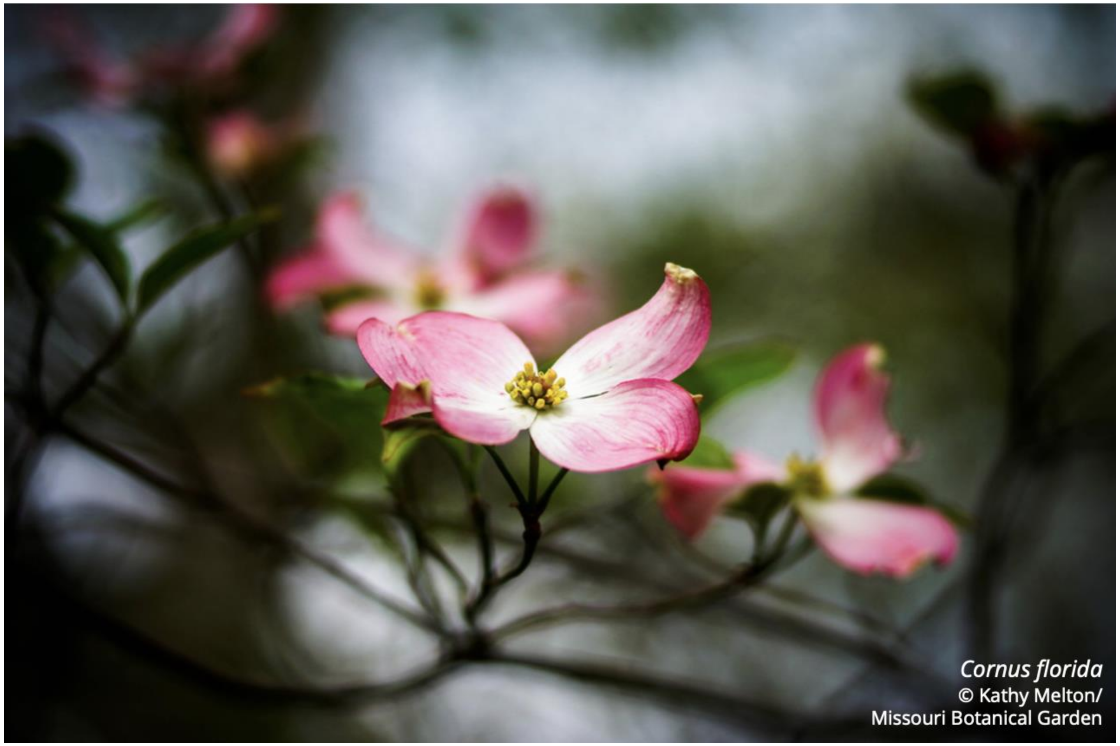
Lecture 13
Python: variables, operators, if-statements, and functions



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

Lecture 13 outline

Last time: phylogenetics

This time: Python (1 of 3)

Python

- variables
- operators
- containers
- functions

https://www.python.org/

Python is a general-purpose scripting language

- ***open source*** language
- ***interpreted*** code is "run-as-read" across platforms
- ***dynamic typing*** of variables
- ***object-oriented*** to allow creation of custom types
- ***high-level*** and symbolic interface with hardware
- ***garbage-collected*** for automatic memory management

Python ecosystem

- ***python***, command line interface *and* scripting program
- ***jupyter***, online python notebooks
- ***pip*** and ***easy_install*** library managers
- ***conda***, python environment emulator
- thousands of libraries
  - scientific computing: *numpy*, *scipy*, *sklearn*
  - datatypes: *pandas*
  - plotting: *matplotlib*, *seaborn*
  - bioinformatics: *biopython*, *etc.*

Python interpreter

open program
from shell

```
$ python

Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # let's get started
>>> print('Hello, world!')
Hello, world!
>>> s = 'Hello, world!'
>>> s
'Hello, world!'
>>>
```

print argument
to standard
output

type variable's
name to view
its value

Variables

Create variables through assignment (=) either directly to values or other variables' values

```
# assign integer (class, 'int')
a = 12
b = a

# assign float (class, 'float')
x = 0.012
y = 1.2E-2
z = x

# assign string (class, 'str')
s = 'twelve'
s = "twelve"
t = s
```

Operators

Produce new values from existing values/variables
Operator behavior depends on data type

```
# declare integer (class, 'int')
a = 12
# declare float (class, 'float')
b = 0.012
# declare string (class, 'str')
c = "12”

# behavior of add operator
a + 1   # 13 (int)
b + 0.1 # 0.112 (float)
c + "1" # "121" (string)
a + b   # 12.012 (float)
a + c   # cannot add int to string
b + c   # cannot add float and string
```

Arithmetic operators take integers/floats as arguments, return an integer/float

```
2 + 2  # addition
2 * 3  # multiplication
7 / 3  # division
9 % 2  # modulus (remainder)
7 // 3 # integer-division
2 ** 3 # exponent
```

Apply an operator then assign the new value to a variable using ***assignment operators***

```
x = 1   # 1, assignment
x += 6  # 7, add-assignment
x -= 3  # 4, subtract-assignment
x *= 2  # 8, multiply-assignment
x /= 4  # 2, division-assignment
x **= 3 # 8, exponent-assignment
x //= 2 # 4, integer-div.-assignment
x %= 3  # 1, modulus-assignment
```

Containers

Containers are variables that store multiple values (often of the same type) called ***elements***

Container elements are generally accessed through the ***index*** operator, *[idx]*

```
>>> # create list called `x`
>>> x = [ 10, 20, 30 ]
>>> # what is the value of `x`?
>>> x
[10, 20, 30]
>>> # access the index-0 element
>>> x[0]
10
```

Lists

List elements are indexed by integers; lists can be modified after creation (*mutable*)

```
>>> x = [ 10, 20, 30 ] # create list called `x`
>>> x # what is the value of `x`?
[10, 20, 30]
>>> x[0] = 11 # set value of index-0 element
>>> x[1:3] = [ 22, 33 ] # set values of index-1,2 elements
>>> x[3] = 55 # access the index-3 element
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> x.append(55) # append element to end of list
>>> x.insert(3, 44) # insert value 44 after 3rd index
>>> x
[11, 22, 33, 44, 55]
```

Dictionaries

Dictionaries contain *key-value* pairs;
keys are used to index values

```
>>> x = { 'a':1, 'b':2 }  # create dictionary with two key-values
>>> x                  # report value of dictionary
{'a': 1, 'b': 2}
>>> x['a’]             # retrieve dictionary value with key 'a'
1
>>> x['c'] = 3         # assign value 3 to key 'c'
>>> x.keys()           # print container of sorted keys
dict_keys(['a', 'b', 'c'])
>>> x.values()         # print container of sorted values
dict_values([1, 2, 3])
>>> x.values()[0]      # dict_values can't be accessed by index!?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object does not support indexing
>>> list(x.values())[0]   # typecast dict_values as list
1
```

Tuples

Tuples are integer-indexed containers, but unlike lists, their contents cannot be modified (*immutable*)

```
>>> x = ( 10, 20, 30 ) # create tuple called `x`
>>> x                  # what is the value of `x`?
(10, 20, 30)
>>> x[0]
10
>>> x[0] = 11          # set value of index-0 element
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Alternatives for initializing lists and

```
>>> x = [] # creates an empty list
>>> x
[]
>>> x = [0]*10 # creates size-10 list with values 0
>>> x
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> x = {} # creates empty dict
>>> x
{}
>>> x = dict.fromkeys(['a','b'], 0) # creates keys with values 0
{'a': 0, 'b': 0}
```

Container elements may differ in type

```
>>> x = [ 1, 0.1 ]
>>> x
[1, 0.1]
>>> type(x)
<class 'list'>
>>> type(x[0])
<class 'int'>
>>> type(x[1])
<class 'float'>
>>> x = [ 'a', [ 1, 0.1, {'a':0, 'b':[0]*2} ], False ]
>>> x
['a', [1, 0.1, {'a': 0, 'b': [0, 0]}], False]
```

Functions

All ***functions*** have a ***name***, and may accept ***parameters***, and may ***return*** a value

```
# declare variables
a = -12
b = 0.012

# print the value of a variable to stdout
print(a)            # returns nothing, but prints to stdout

# learn the type of a variable
s = type(b)         # returns string with value '<class str>'

# convert a variable into a string
x = str(a + b)      # returns string with value '-11.988'

# get the absolute value of a number
y = abs(a)          # returns integer with value 12

# round a number
z = round(b, ndigits=2)  # returns float with value 0.01

# nested functions, evaluated in order of
# innermost to outermost function call
print(abs(round(a+b, ndigits=2)))
```

Writing a custom function

All ***functions*** have a ***name***, and may accept ***parameters***, and may ***return*** a value

```
# my function
def add_three(x1, x2, x3=5):
   y = x1 + x2 + x3
   return(y)

# assign return value to variable
a = add_three(2.0203, -1, 2.3)
b = round(a, ndigits=2)
print(b)

# pass return value as a parameter
# `add_three()` not provided third argument,
# so it uses `x3=5` as default
print(round(add_three(2.0203, -1), ndigits=2))

# why does this create an error?
print(add_three(1, 2, '3'))
```

Code blocks and whitespace

Programming languages often use ***code blocks*** to define the ***scope*** for complex constructs – *e.g. functions, if-statements, for-loops, classes*

Python uses *aligned* ***whitespace*** *identations* to define code blocks. This results in code that is cleaner and easier to read, but that is very sensitive to formatting.

```
# valid block, aligned to 4 spaces
def f(x):
    x += 1
    print('increment')
    return x

# invalid block, misaligned
def f(x):
     x += 1
   print('increment')
    return x
```

Help function

Call *help()* retrieves information on use for functions, classes, methods, modules, etc.

```
>>> # What does the `print()` function do,
>>> # and how is it used?
>>> help(print)

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

Help function

The *help()* function can target functions with ***docstrings***;
Docstrings are enclosed by triple quotes (''') and appear in the next line(s) after the function definition

```
>>> def my_function(arg1):
... '''
... This function raises a number by the power of itself.
...
... Parameters:
... arg1 (int): the number
...
... Returns:
... int: arg1 raised to the power arg1
... '''
...
... return arg1**arg1
...
>>> help(my_function)
Help on function my_function in module __main__:

my_function(arg1)
    This function raises a number by the power of itself

    Parameters:
    arg1 (int): the number

    Returns:
    int: arg1 raised to the power arg1
(END)
```

Running Python scripts

```
$ # view contents of Python script
$ cat example.py
#!/bin/python

# this will print to stdout
print('Hello, world!’)

# this only prints to python interface
s = ‘Hey, planet...’
s

$ # supply `python` with script as argument
$ python example.py
Hello, world!

$ # set script as executable by `#!/bin/python`
$ chmod +x example.py
$ ./example.py
Hello, world!
```

provide script name

...or execute script

System arguments

Python programs can accept user arguments through the system argument vector, *sys.argv*

```
# load system library
import sys

# print each argument
print('sys.argv contents:')
for i,v in enumerate(sys.argv):
   print(' ' + str(i) + ' : ' + str(v))

# done
print('...done!')
```

Pass arguments to Python when calling the script from shell

```
$ python example.py 10 20
sys.argv contents:
 0 : example.py
 1 : 10
 2 : 20
...done!
```

Writing pythonic code

```
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Overview for Lab 13