Lecture 12
Python: variables, operators, if-statements, and functions

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

Lecture 12 outline

Last time: phylogenetics

This time: Python (1 of 4)

Python

- variables
- operators
- if-statements
- functions

python™

https://www.python.org/

Python is a general-purpose scripting language

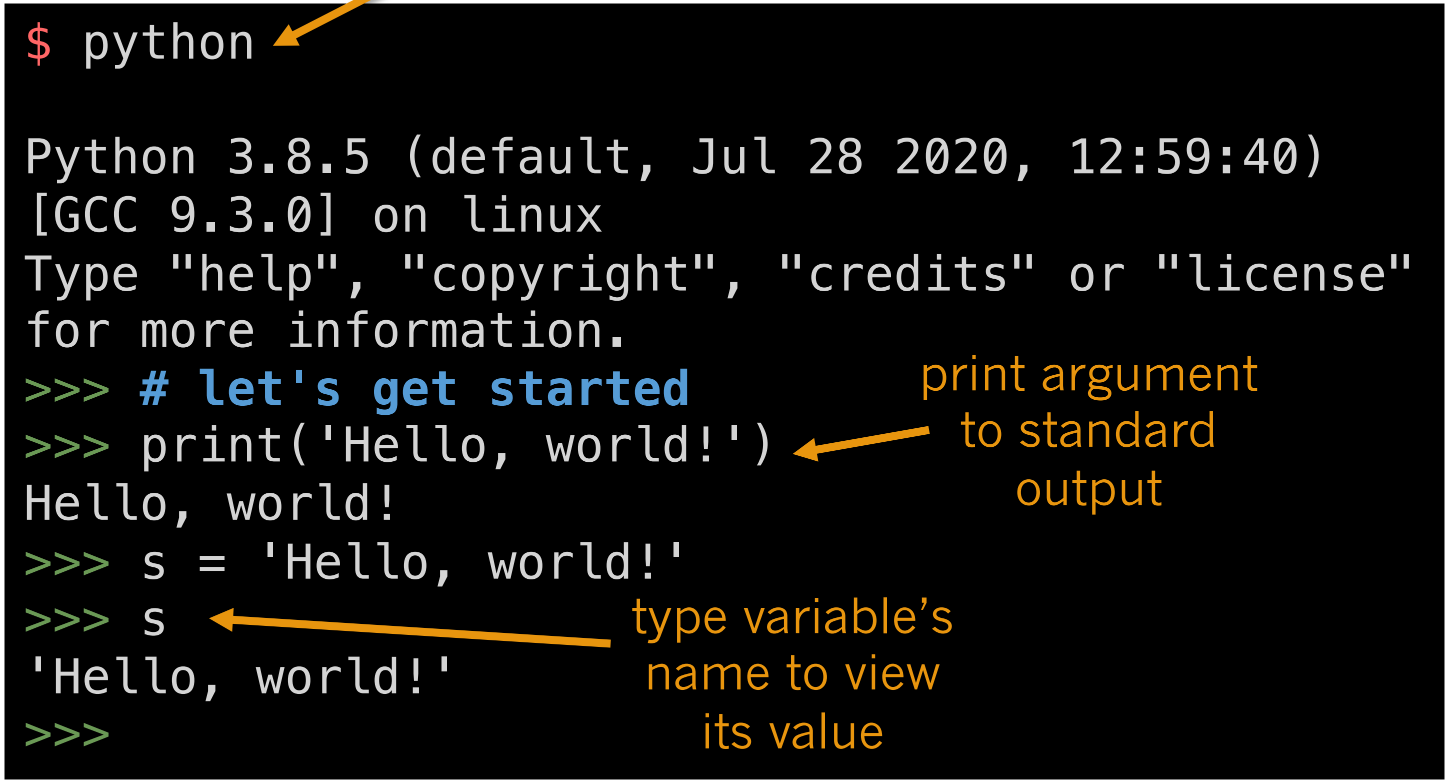
- ***open source*** language
- ***interpreted*** code is "run-as-read" across platforms
- ***dynamic typing*** of variables
- ***object-oriented*** to allow creation of custom types
- ***high-level*** and symbolic interface with hardware
- ***garbage-collected*** for automatic memory management

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



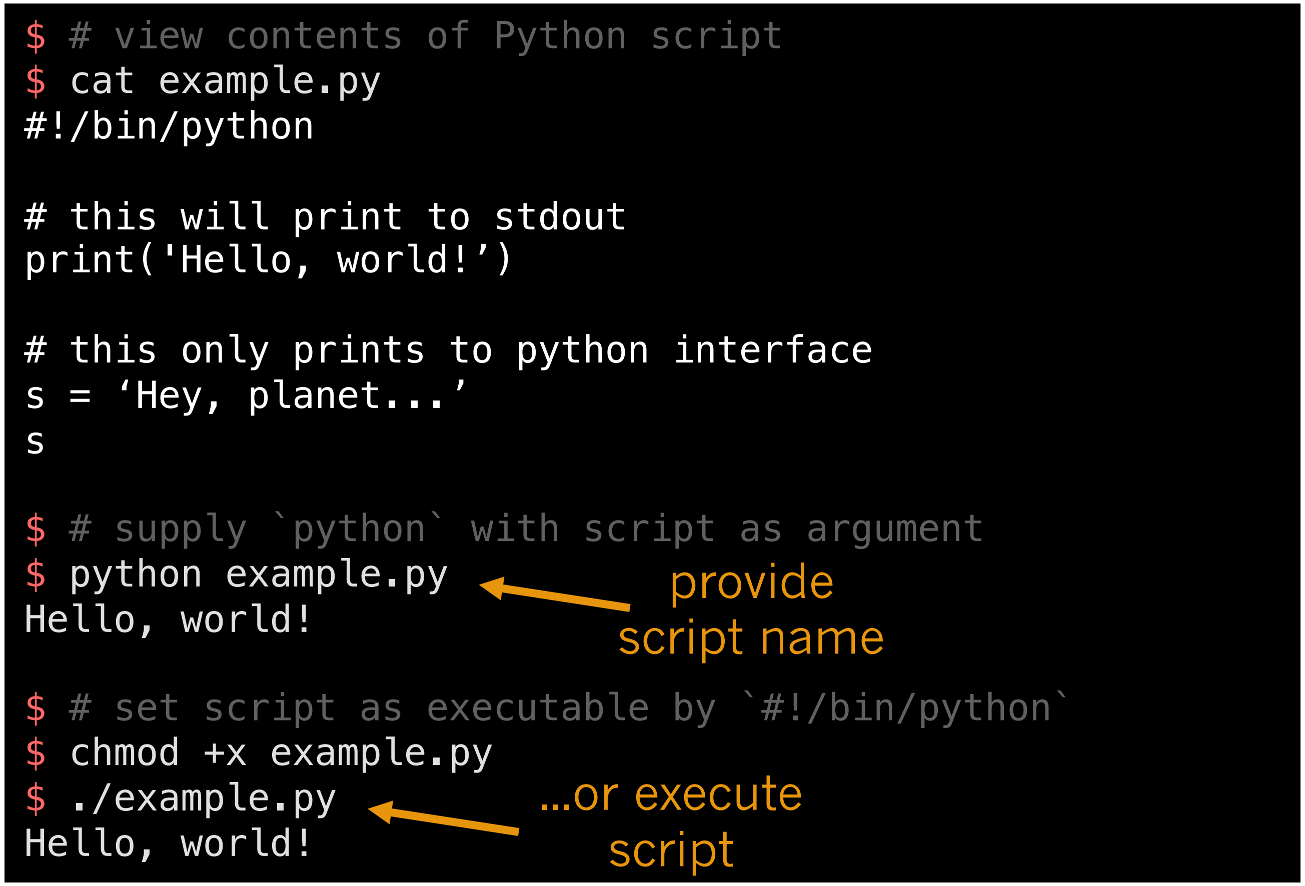
```
$ python

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

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!
```

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
```

Boolean operators return True if condition(s) are met and False otherwise

```
7 == 6         # is-equal                (False)
7 != 6         # is-not equal            (True)
9 < 3          # less-than               (False)
8 <= 9         # less-than-or-equal      (True)
7 > 6          # greater-than            (True)
4 >= 5         # greater-than-or-equal   (False)

1 < 3 and 3<2  # AND operator            (False)
1<3 or 3<2     # OR operator             (True)
not 3<2        # NOT operator            (True)
```

Operator precedence

Operations are evaluated in their order of precedence

```
# operator precedence (high to low)
(...), [...], {key: value}, {...}    # 1. groups, tuples, lists, dict., sets
x[index], f(arguments), x.attribute  # 2. containers, functions, objects
**                                   # 3. exponent
-x                                   # 4. negation
*, /, //, %                          # 5. multiply/division
+, -                                 # 6. addition/substraction
<, <=, >, >=, !=, ==                 # 7. comparisons
is, not, in, is, is not              #    (cont'd)
not x                                # 8. boolean not
and                                  # 9. boolean and
or                                   # 10. boolean or
```

Use parentheses to adjust precedence

```
5 * 2 + 4 * 3             # 22
(5 * 2) + (4 * 3)         # 22 (same precedence)
(5 * 2 + 4) * 3           # 42
5 * (2 + 4 * 3)           # 70
5 * ((2 + 4) * 3)         # 90
```

Combining operators

Are the following comparisons True or False?
Solve by hand.

```
# create variables
a = 1
b = 3
c = 2.1

# True or False?
a + b < c * 2                                   # comparison 1
b + c - 0.1 >= 3 * c                            # comparison 2
b * (a + c) < b**2 + (a / 10)                   # comparison 3
b % c > ((c + a) > b) or ((2*a) > c)            # comparison 4
```

if-statements

Executes code block only when
if, *elif*, and *else* conditions are met

```
a = 1
b = 2

# execute each code block if condition is True
if a == b:
    # if a equals b
    print('a is equal to b’)
    b += 1
elif a < b:
    # else if a less than b
    print('a is less than b’)
    a -= 1
else:
    # otherwise, if a does not equal b
    # and if a is not less than b
    print('a is greater than b’)
    a *= b

c = a + b
print(c)
```

Code blocks and whitespace

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

```python
# valid block, aligned to 4 spaces
if x < 0:
    x += 1
    print('increment')

# invalid block, misaligned
if x < 0:
     x += 1
   print('increment')
```

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)))
```

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)
```

Writing a custom function

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

```python
# 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'))
```

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

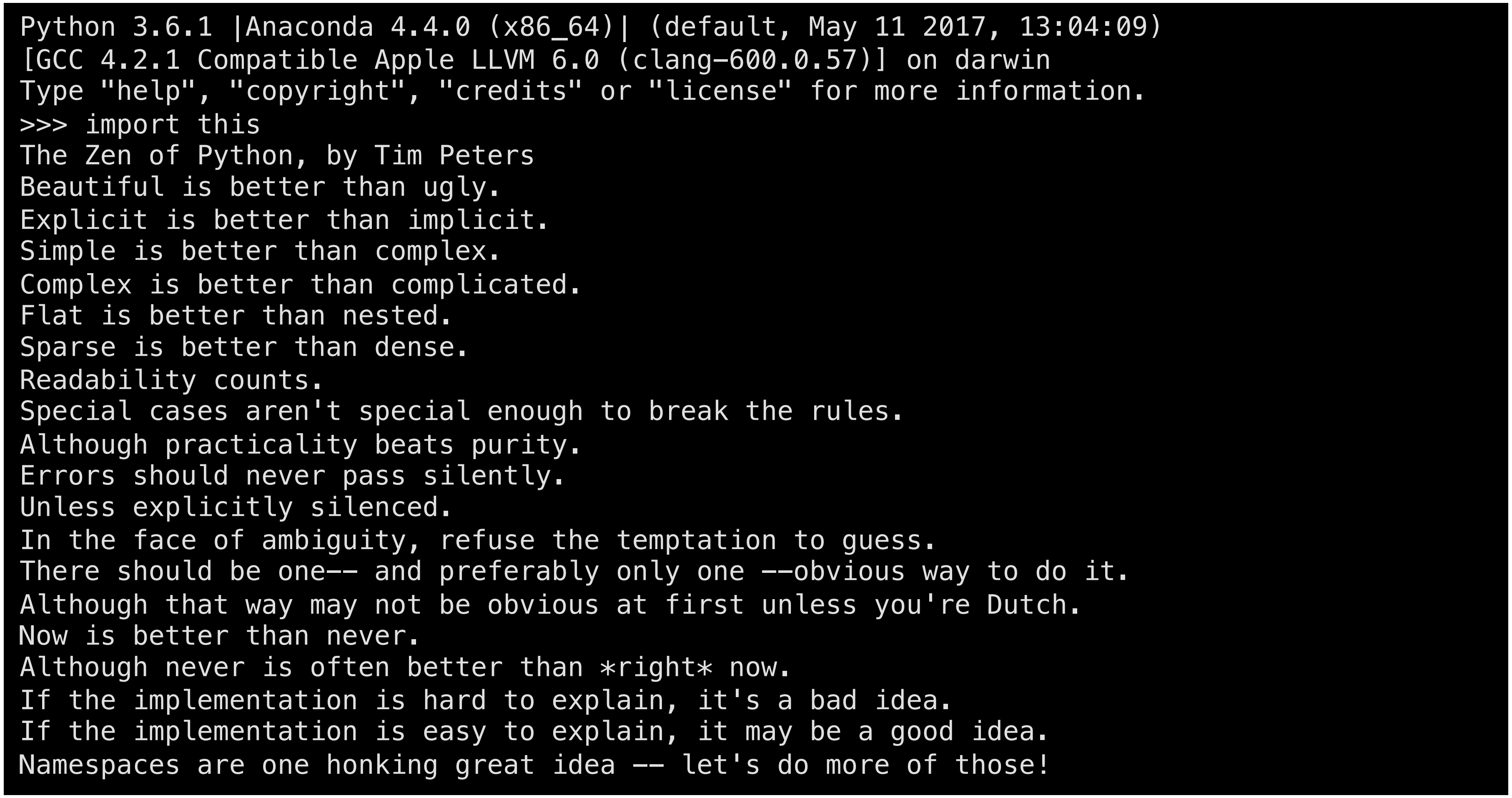
```
>>> def my_function(arg1):
... '''
... This function raises a number by the power of itself.
...
... Parameters:
... arg1 (int): the number
...
... Returns:
... int: arg1 raised to the power arg1
... '''
...
... return arg1**arg1
...
>>> help(my_function)
Help on function my_function in module __main__:

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)
```

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 12