Algorithmic Approaches for Biological Data, Lecture #2

Katherine St. John

City University of New York
American Museum of Natural History

25 January 2016

Outline

- Python 2 vs. 3

Outline

- Python 2 vs. 3
- Python variables & arithmetic

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries
- More on `for` and `range()`

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries
- More on `for` and `range()`
- Break

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries
- More on `for` and `range()`
- Break
- Accumulator Design Pattern: Using variables as accumulators

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries
- More on `for` and `range()`
- Break
- Accumulator Design Pattern: Using variables as accumulators
- Input-Process-Output (IPO) design pattern

Outline

- Python 2 vs. 3
- Python variables & arithmetic
- Using Python Modules: turtle, math & random libraries
- More on `for` and `range()`
- Break
- Accumulator Design Pattern: Using variables as accumulators
- Input-Process-Output (IPO) design pattern
- Introduction to Functions

Python 2 vs. 3

- More specifically, Python 2.7 or 3.5?

Python 2 vs. 3

- More specifically, Python 2.7 or 3.5?
- Python 3 (2008) is very stable, fixes many oddities, but not backwards-compatible

Python 2 vs. 3

- More specifically, Python 2.7 or 3.5?
- Python 3 (2008) is very stable, fixes many oddities, but not backwards-compatible
- Python 2 is no longer actively developed (final version was in 2010).

Python 2 vs. 3

- More specifically, Python 2.7 or 3.5?
- Python 3 (2008) is very stable, fixes many oddities, but not backwards-compatible
- Python 2 is no longer actively developed (final version was in 2010).
- Many popular packages (as well as Mac OSX) still run Python 2.x.

Python 2 vs. 3

- More specifically, Python 2.7 or 3.5?
- Python 3 (2008) is very stable, fixes many oddities, but not backwards-compatible
- Python 2 is no longer actively developed (final version was in 2010).
- Many popular packages (as well as Mac OSX) still run Python 2.x.
- We will use Python 2 but will point out the differences. Such as:

Python 2	Python 3
`print "Hello!"`	`print("Hello!")`

Data Types

- Information can be stored in variables.

Data Types

- Information can be stored in variables.

```
x = 5
x = x + 1
firstName = "Teddy"
ave = total/ 10.0
```

Data Types

- Information can be stored in variables.

```
x = 5
x = x + 1
firstName = "Teddy"
ave = total/ 10.0
```

- Some basic types: integers, floating point (real numbers), strings (sequences of characters).

Data Types

- Information can be stored in variables.

```
x = 5
x = x + 1
firstName = "Teddy"
ave = total/ 10.0
```

- Some basic types: integers, floating point (real numbers), strings (sequences of characters).
- Don't need to say what kind of variable– Python decides from context ("dynamic typing").

Data Types

- Information can be stored in variables.

```
x = 5
x = x + 1
firstName = "Teddy"
ave = total/ 10.0
```

- Some basic types: integers, floating point (real numbers), strings (sequences of characters).
- Don't need to say what kind of variable– Python decides from context ("dynamic typing").

`PythonTutor` *Demo: variable assignments, incrementing variables, simultaneous assignment*

Group Work

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1. ```
   y = 5
   x = y*2
   ```

2. ```
   a = 1
   b = "hi"
   a, b = b, a
   ```

3. ```
   first = 0
   second = 2**first
   third = second % 5
   ```

4. ```
   cake = 10
   people = 3
   slice1 = cake / people
   slice2 = cake / float(people)
   ```

5. ```
   t = "Teddy"
   r = "Roosevelt"
   name = t + " " + r
   ```
   ```

Python Modules

- Libraries (packages) of useful functions.

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`
  - Contains useful math functions and constants such as:

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`
  - Contains useful math functions and constants such as:
    π `math.pi`

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`
  - Contains useful math functions and constants such as:

π	`math.pi`
$\cos(x)$	`math.cos(x)`

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`
  - Contains useful math functions and constants such as:

π	`math.pi`
$\cos(x)$	`math.cos(x)`
$\log_b(x)$	`math.log(x, b)`

Python Modules

- Libraries (packages) of useful functions.
- Already used `turtle` and `random` modules
- Another built-in module is `math`:
  - To use, `import math`
  - Contains useful math functions and constants such as:

π	`math.pi`
$\cos(x)$	`math.cos(x)`
$\log_b(x)$	`math.log(x, b)`

- Will see other modules: `numpy`, `matplotlib`, and `scipy` (all free, but be downloaded & installed)

`PythonTutor` *Demo*

for and range

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

for and range

- An example of a definite loop.

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

for and range

- An example of a definite loop.
- The body of the loop is the statements indented after it.

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

for and range

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

- An example of a definite loop.
- The body of the loop is the statements indented after it.
- x is called the loop variable or index variable.

for and range

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

- An example of a definite loop.
- The body of the loop is the statements indented after it.
- `x` is called the loop variable or index variable.
- `range(4)` creates a list `[0,1,2,3]`.

for and range

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

- An example of a definite loop.
- The body of the loop is the statements indented after it.
- `x` is called the loop variable or index variable.
- `range(4)` creates a list `[0,1,2,3]`.
- Roughly, the `for` loop:
  1. assigns next value of list to `x`,
  2. does the body statements, and
  3. then if there's still list items goes back to #1;
     else ends loop.

for and range

```
for x in range(4):
    tess.forward(10)
    tess.right(90)
```

- An example of a definite loop.
- The body of the loop is the statements indented after it.
- `x` is called the loop variable or index variable.
- `range(4)` creates a list `[0,1,2,3]`.
- Roughly, the `for` loop:
  1. assigns next value of list to `x`,
  2. does the body statements, and
  3. then if there's still list items goes back to #1;
     else ends loop.
- *PythonTutor demo*

range-statements

- `range()` returns a list of numbers.

Standard form:
`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

range-statements

- `range()` returns a list of numbers.
- Simplest: `range(stop)`
  returns `[0,1,2,...stop-1]`

Standard form:
`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

range-statements

Standard form:
`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

- `range()` returns a list of numbers.
- Simplest: `range(stop)`
  returns `[0,1,2,...stop-1]`
- `range(start, stop)`
  returns
  `[start,start+1,...stop-1]`

range-statements

Standard form:
`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

- `range()` returns a list of numbers.
- Simplest: `range(stop)`
  returns `[0,1,2,...stop-1]`
- `range(start, stop)`
  returns
  `[start,start+1,...stop-1]`
- `range(start, stop,step)`
  returns
  `[start,start+step,start+2*step,...,U]`
  where U is largest `start +k*step`
  that is less than `stop`.

range-statements

Standard form:
`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

- `range()` returns a list of numbers.
- Simplest: `range(stop)`
  returns `[0,1,2,...stop-1]`
- `range(start, stop)`
  returns
  `[start,start+1,...stop-1]`
- `range(start, stop,step)`
  returns
  `[start,start+step,start+2*step,...,U]`
  where U is largest `start +k*step`
  that is less than `stop`.
- *PythonTutor demo*

Group Work

Standard form:

`for x in <list>:`
`  command1`
`  command2`
`  ...`
`  commandN`

Standard form:

`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1.
```
for num in range(5):
    print num*num
```

Group Work

Standard form:
`for x in <list>:` command1 command2 ... commandN

Standard form:

`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1.
```
for num in range(5):
    print num*num
```
2.
```
for e in range(2,20,2):
    print e
```

Group Work

Standard form:

`for x in <list>:`
`  command1`
`  command2`
`  ...`
`  commandN`

Standard form:

`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1.
```
for num in range(5):
    print num*num
```
2.
```
for e in range(2,20,2):
    print e
```
3.
```
for b in range(10,0,-1):
    print b
print "Blast off!"
```

Group Work

Standard form:
`for x in <list>:` command1 command2 ... commandN

Standard form:

`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1.
```
for num in range(5):
    print num*num
```
2.
```
for e in range(2,20,2):
    print e
```
3.
```
for b in range(10,0,-1):
    print b
print "Blast off!"
```
4.
```
for x in [3,1,4,1,5,9]:
    print x,
```

Group Work

Standard form:
`for x in <list>:` command1 command2 ... commandN

Standard form:

`range(stop)`
`range(start,stop)`
`range(start, stop, step)`

In pairs (or triplets), work out the following (and then try at the Python shell or pythonTutor):

1. ```
   for num in range(5):
       print num*num
   ```
2. ```
   for e in range(2,20,2):
       print e
   ```
3. ```
   for b in range(10,0,-1):
       print b
   print "Blast off!"
   ```
4. ```
   for x in [3,1,4,1,5,9]:
       print x,
   ```
5. ```
   tess = turtle.Turtle()
   for c in ["blue","red","green","yellow"]:
       tess.color(c)
       tess.forward(50)
       tess.right(90)
   ```
   ```

Group Work

Standard form:
for x in <list>: command1 command2 ... commandN

Standard form:
range(stop) range(start,stop) range(start, stop, step)

In pairs (or triplets), work out the following (and then try at the Python shell or pythonTutor):

1. ```
   for num in range(5):
       print num*num
   ```
2. ```
   for e in range(2,20,2):
       print e
   ```
3. ```
   for b in range(10,0,-1):
       print b
   print "Blast off!"
   ```
4. ```
   for x in [3,1,4,1,5,9]:
       print x,
   ```
5. ```
   tess = turtle.Turtle()
   for c in ["blue","red","green","yellow"]:
       tess.color(c)
       tess.forward(50)
       tess.right(90)
   ```
6. ```
   total = 0
   for i in range(11,22,2):
       total = total + i
       print i, total
   ```

Break

Accumulator Design Pattern

- "Accumulating" a value is a design approach that occurs frequently.

Accumulator Design Pattern

- "Accumulating" a value is a design approach that occurs frequently.
- Example:

```
total = 0
for i in range(11,22,2):
    total = total + i
    print i, total
```

- Simplest forms are running sums and running products.

Accumulator Design Pattern

- "Accumulating" a value is a design approach that occurs frequently.
- Example:

```
total = 0
for i in range(11,22,2):
    total = total + i
    print i, total
```

- Simplest forms are running sums and running products.

	Running Sums:	Running Products:	Accumulating Strings:	Accumulating Lists:
Initialization:	s = 0	p = 1	Explained	Explained
Update Action:	for ... s=s+newValue	for ... p=p*newValue	Later	Later

Group Work

In pairs (or triplets), work out the following
(and then try at the Python shell or `pythonTutor`):

Group Work

In pairs (or triplets), work out the following
(and then try at the Python shell or `pythonTutor`):

1. ```
   fact = 1
   for i in range(10,1,-1):
       fact = fact * i
       print i, fact
   ```
   ```

Group Work

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1. ```
   fact = 1
   for i in range(10,1,-1):
       fact = fact * i
       print i, fact
   ```

2. ```
   f = 0
   g = 1
   for i in range(10):
       t = f + g
       f = g
       g = t
       print i, f
   ```

Group Work

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1. ```
   fact = 1
   for i in range(10,1,-1):
       fact = fact * i
       print i, fact
   ```

2. ```
   f = 0
   g = 1
   for i in range(10):
       t = f + g
       f = g
       g = t
       print i, f
   ```

3. Write code that sums up the first 100 odd numbers.

Group Work

In pairs (or triplets), work out the following (and then try at the Python shell or `pythonTutor`):

1.
```
fact = 1
for i in range(10,1,-1):
    fact = fact * i
    print i, fact
```

2.
```
f = 0
g = 1
for i in range(10):
    t = f + g
    f = g
    g = t
    print i, f
```

3. Write code that sums up the first 100 odd numbers.

4. Write code that calculates the double factorial: $1!! = 1$, $2!! = 2$, $3!! = 3 \cdot 1!!$, $4!! = 4 \cdots 2!!$. Pattern: $n!! = n \cdot (n-2)!!$

Input-Process-Output Pattern

- Very common design pattern.

Input-Process-Output Pattern

- Very common design pattern.
- Form is:

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation
  - **Process:** operations performed on the inputs

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation
  - **Process:** operations performed on the inputs
  - **Output:** what's printed/returned at the end

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation
  - **Process:** operations performed on the inputs
  - **Output:** what's printed/returned at the end
- Example:
  - **Input:** Amount of snow in inches

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation
  - **Process:** operations performed on the inputs
  - **Output:** what's printed/returned at the end
- Example:
  - **Input:** Amount of snow in inches
  - **Process:** Multiply input by 2.54

Input-Process-Output Pattern

- Very common design pattern.
- Form is:
  - **Input:** what's given to the computation
  - **Process:** operations performed on the inputs
  - **Output:** what's printed/returned at the end
- Example:
  - **Input:** Amount of snow in inches
  - **Process:** Multiply input by 2.54
  - **Output:** Equivalent amount in centimeters

Functions

Functions:

- **Input parameters:** information that goes into the function

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function
- **Return values:** information sent back by the function

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function
- **Return values:** information sent back by the function

Example: doubling function

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function
- **Return values:** information sent back by the function

Example: doubling function

```
def doubling(x):
```

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function
- **Return values:** information sent back by the function

Example: doubling function

```
def doubling(x):
    d = 2*x
```

Functions

Functions:

- **Input parameters:** information that goes into the function
- **Body statements:** commands inside the function
- **Return values:** information sent back by the function

Example: doubling function

```
def doubling(x):
    d = 2*x
    return d
```

In Pairs

Write function definitions for:

- Function that returns the square of the input

Standard form:

`def myFunc(in1,in2,...):`
`  command1`
`  command2`
`  ...`
`  return(out1,out2,...)`

In Pairs

Standard form:

`def myFunc(in1,in2,...):`
`  command1`
`  command2`
`  ...`
`  return(out1,out2,...)`

Write function definitions for:

- Function that returns the square of the input
- Function that returns 1/10 of the input

In Pairs

Standard form:
`def myFunc(in1,in2,...):` command1 command2 ... return(out1,out2,...)

Write function definitions for:

- Function that returns the square of the input
- Function that returns 1/10 of the input
- Function that takes two inputs and returns the sum

In Pairs

Standard form:
`def myFunc(in1,in2,...):` command1 command2 ... return(out1,out2,...)

Write function definitions for:

- Function that returns the square of the input
- Function that returns 1/10 of the input
- Function that takes two inputs and returns the sum
- Function that takes two inputs and returns the product

In Pairs

Standard form:
`def myFunc(in1,in2,...):` command1 command2 ... return(out1,out2,...)

Write function definitions for:

- Function that returns the square of the input
- Function that returns 1/10 of the input
- Function that takes two inputs and returns the sum
- Function that takes two inputs and returns the product
- Function that takes two inputs and returns them in reverse order

Recap

- More on functions on Wednesday, 1pm.

Recap

- More on functions on Wednesday, 1pm.
- Email lab reports to `kstjohn@amnh.org`

Recap

- More on functions on Wednesday, 1pm.
- Email lab reports to `kstjohn@amnh.org`
- Challenges available at `rosalind.info` (use emailed link to access course page).