Algorithmic Approaches for Biological Data, Lecture #4

Katherine St. John

City University of New York
American Museum of Natural History

1 February 2016

Outline

- Review: loops, functions

Outline

- Review: loops, functions
- Simple input methods

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators
- Decisions

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators
- Decisions
- Break

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators
- Decisions
- Break
- Nested Statements

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators
- Decisions
- Break
- Nested Statements
- Conditional Execution

Outline

- Review: loops, functions
- Simple input methods
- Boolean values & Expressions
- Precedence of Operators
- Decisions
- Break
- Nested Statements
- Conditional Execution
- Using Decisions in Programs

Review: loops

Standard form:

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

Roughly, the `for` loop:

Standard form:

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

Review: loops

Standard form:

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

Standard form:

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

Roughly, the `for` loop:

1. assigns next value of list to `x`,

Review: loops

Standard form:

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

Standard form:

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

Roughly, the `for` loop:

1. assigns next value of list to `x`,
2. does the body statements, and

Review: loops

Standard form:

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

Standard form:

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

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.

Review: functions

Functions:

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

Standard form:

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

Review: functions

Standard form:

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

Functions:

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

Review: functions

Standard form:

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

Functions:

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

Review: functions

Standard form:

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

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

Review: functions

Standard form:

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

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):
```

Review: functions

Standard form:

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

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
```

Review: functions

Standard form:

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

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
```

Group Work

In pairs/triples, work out (and then try at the shell or `pythonTutor`):

1. ```
   for num in range(3,10,2):
       print num
   ```
2. ```
   for i in range(5):
       for j in range(5):
           print j,
       print
   ```
3. ```
   for i in range(5):
       for j in range(5):
           print i,
       print
   ```
4. ```
   for i in range(5):
       for j in range(i):
           print "*",
       print
   ```
5. ```
   for s in ["a","m","n","h"]:
       print s,
   ```
6. ```
   for s in "amnh":
       print s,
   ```
7. ```
   total = 0
   for i in range(100,201,10):
       total = total + i
       print i, total
   print "Final total is", total
   ```
8. ```
   def mys(x,y):
       t = x*y
       return(t)
   print mys(2,5), mys(5,2)
   ```
9. ```
   def compute(x,y):
       return x-y, x+y
   for i in range(5):
       print compute(10,i)
   ```
   ```

Simple input methods

In Python 2:

- To get strings (sequences of characters):

```
name = raw_input('Please enter your name:  ')
```

Simple input methods

In Python 2:

- To get strings (sequences of characters):
  `name = raw_input('Please enter your name:  ')`
- To get numbers (evaluates the string):
  `num = input('Please enter a number:  ')`

Simple input methods

In Python 2:

- To get strings (sequences of characters):
  `name = raw_input('Please enter your name:  ')`
- To get numbers (evaluates the string):
  `num = input('Please enter a number:  ')`
- Can also get multiple numbers:
  `hrs, mins = input('Please enter hours and minutes, separated by a comma:  ')`

Simple input methods

In Python 2:

- To get strings (sequences of characters):
  `name = raw_input('Please enter your name: ')`
- To get numbers (evaluates the string):
  `num = input('Please enter a number: ')`
- Can also get multiple numbers:
  `hrs, mins = input('Please enter hours and minutes, separated by a comma: ')`
- `input()` takes the user's input and 'evaluates' it– that is, simplifies it as much as possible:

Simple input methods

In Python 2:

- To get strings (sequences of characters):
  `name = raw_input('Please enter your name:  ')`
- To get numbers (evaluates the string):
  `num = input('Please enter a number:  ')`
- Can also get multiple numbers:
  `hrs, mins = input('Please enter hours and minutes, separated by a comma:  ')`
- `input()` takes the user's input and 'evaluates' it– that is, simplifies it as much as possible:

```
>>> num = input('Please enter a number:  ')
>>> print num
```

Simple input methods

In Python 2:

- To get strings (sequences of characters):
  ```
  name = raw_input('Please enter your name:  ')
  ```
- To get numbers (evaluates the string):
  ```
  num = input('Please enter a number:  ')
  ```
- Can also get multiple numbers:
  ```
  hrs, mins = input('Please enter hours and
  minutes, separated by a comma:  ')
  ```
- `input()` takes the user's input and 'evaluates' it– that is, simplifies it as much as possible:
  ```
  >>> num = input('Please enter a number:  ')
  >>> print num
  Please enter a number:  2**5
  32
  ```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

  `False` `True`

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

- Examples at Python shell:

```
print(3>4)
```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

- Examples at Python shell:

```
print(3>4)
print(0!=1)
```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

- Examples at Python shell:

```
print(3>4)
print(0!=1)
x = 5
print(x*x == 25)
```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

- Examples at Python shell:

```
print(3>4)
print(0!=1)
x = 5
print(x*x == 25)
print(x*x == x**2)
```

Boolean values & Expressions

In Python 2:

- Primitive Data types: int, float, string
- Boolean: represents values that are true or false
- Python writes these values with capital letters:

```
False   True
  0       1
```

- Examples at Python shell:

```
print(3>4)
print(0!=1)
x = 5
print(x*x == 25)
print(x*x == x**2)
```

- Useful for making decisions:
  If the number is positive ....

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

```
print (not 0==1)
```

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

```
print (not 0==1)
print (not (False and True)) x = 21
```

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

```
print (not 0==1)
print (not (False and True)) x = 21
print(x % 2 == 0 or x % 3 == 0)
```

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

```
print (not 0==1)
print (not (False and True)) x = 21
print(x % 2 == 0 or x % 3 == 0)
print(x < 0 and x > 100)
```

Boolean values & Expressions

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

- Can join together boolean expressions:
  - `and`: true when both statements are true
  - `or`: false when both statements are false
  - `not`: toggles truth value
- Examples at Python shell:

```
print (not 0==1)
print (not (False and True)) x = 21
print(x % 2 == 0 or x % 3 == 0)
print(x < 0 and x > 100)
```

Precedence of Operators

The precedence, or priority, that operators are performed follows the rules from mathematics:

Precedence of Operators

The precedence, or priority, that operators are performed follows the rules from mathematics:

5.15. Operator precedence

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section Comparisons — and exponentiation, which groups from right to left).

Operator	Description
`lambda`	Lambda expression
`if – else`	Conditional expression
`or`	Boolean OR
`and`	Boolean AND
`not x`	Boolean NOT
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, including membership tests and identity tests
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<, >>`	Shifts
`+, -`	Addition and subtraction
`*, /, //, %`	Multiplication, division, remainder [8]
`+x, -x, ~x`	Positive, negative, bitwise NOT
`**`	Exponentiation [9]
`x[index], x[index:index], x(arguments...), x.attribute`	Subscription, slicing, call, attribute reference
`(expressions...), [expressions...], {key: value...}, `expressions...``	Binding or tuple display, list display, dictionary display, string conversion

Decisions

Standard form:

```
if <test1>:
    command
    command
    ...
elif <test2>:
    command
    command
    ...
else:
    command
    command
    ...
```

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
elif x % 3 == 0:
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
elif x % 3 == 0:
    print x, "is odd and divisible by 3"
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
elif x % 3 == 0:
    print x, "is odd and divisible by 3"
else:
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
elif x % 3 == 0:
    print x, "is odd and divisible by 3"
else:
    print x, "isn't divisible by 2 or 3"
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- **Conditional Tests:** if true, do first block, otherwise go to next test/block
- `if` **block:** performed if test is true
- `elif` **test:** checked if above test1 is false
- `elif` **block:** performed if `elif` test2 is true
- `else` **block:** performed if `elif` test2 is false

Example:

```
if x % 2 == 0:
    print x, "is even"
elif x % 3 == 0:
    print x, "is odd and divisible by 3"
else:
    print x, "isn't divisible by 2 or 3"
```

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- Tests should be expressions that evaluate to `True` or `False`.

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- Tests should be expressions that evaluate to `True` or `False`.
- Can add in extra `elif`'s for more tests.

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- Tests should be expressions that evaluate to `True` or `False`.
- Can add in extra `elif`'s for more tests.
- Can leave off `elif` if no need for a second test.

Decisions

Standard form:

`if <test1>:`
  command
  command
  ...
elif <test2>:
  command
  command
  ...
else:
  command
  command
  ...

Decisions:

- Tests should be expressions that evaluate to `True` or `False`.
- Can add in extra `elif`'s for more tests.
- Can leave off `elif` if no need for a second test.
- Can leave off `else` if no need to do something if the test is false.

Group Work

In pairs/triples, work out (and then try at the shell or `pythonTutor`):

1.
```
if num > 0:
    print "positive!"
elif num < 0:
    print "negative"
else:
    print "zero"
```

2.
```
if num > 0:
    print "positive!"
elif num < 0:
    print "negative"
else:
    print "zero"
```

3.
```
if name != "":
    print "Name is", name
else:
    print "No name entered"
```

4.
```
if n < 0 or n > 100:
    print "Value Out of Range!"
```

5.
```
if 0 < n < 100 and n*n >50:
    print "yes"
```

6.
```
for num in range(10):
    if num % 2 == 0:
        print num
```

7.
```
for num in range(-5,5):
    if num >= 0:
        print num
    else:
        print "-"
```

8.
```
for i in range(5):
    for j in range(5):
        if (i+j) % 2 == 0:
            print "+",
        else:
            print "-",
    print
```

Recap

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

Standard form:

```
if <test1>:
    command
    command
    ...
elif <test2>:
    command
    command
    ...
else:
    command
    command
    ...
```

- Boolean expressions and variables have values `True` or `False`.

Recap

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

Standard form:

```
if <test1>:
    command
    command
    ...
elif <test2>:
    command
    command
    ...
else:
    command
    command
    ...
```

- Boolean expressions and variables have values `True` or `False`.
- Used as the test for decisions (as well as indefinite loops (coming soon!))

Recap

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

Standard form:

```
if <test1>:
    command
    command
    ...
elif <test2>:
    command
    command
    ...
else:
    command
    command
    ...
```

- Boolean expressions and variables have values `True` or `False`.
- Used as the test for decisions (as well as indefinite loops (coming soon!))
- `if` statements can have multiple clauses.

Recap

P	Q	P and Q
F	F	F
F	T	F
T	F	F
T	T	T

P	Q	P or Q
F	F	F
F	T	T
T	F	T
T	T	T

P	not P
F	T
T	F

Standard form:

```
if <test1>:
    command
    command
    ...
elif <test2>:
    command
    command
    ...
else:
    command
    command
    ...
```

- Boolean expressions and variables have values `True` or `False`.
- Used as the test for decisions (as well as indefinite loops (coming soon!))
- `if` statements can have multiple clauses.

Break

Nested Statements

- As we saw, loops can be nested inside loops:

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

- And if-statements can be nested inside loops:

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

- And if-statements can be nested inside loops:

```
for i in range(5):
    for j in range(5):
        if i+j % 2 == 0:
            print "+",
        else:
            print "-",
    print
```

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

- And if-statements can be nested inside loops:

```
for i in range(5):
    for j in range(5):
        if i+j % 2 == 0:
            print "+",
        else:
            print "-",
    print
```

- As long as you indent correctly, you can nest loops and decisions.

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

- And if-statements can be nested inside loops:

```
for i in range(5):
    for j in range(5):
        if i+j % 2 == 0:
            print "+",
        else:
            print "-",
    print
```

- As long as you indent correctly, you can nest loops and decisions.
- *Think CS* examples

Nested Statements

- As we saw, loops can be nested inside loops:

```
for i in range(5):
    for j in range(i):
        print "*",
    print
```

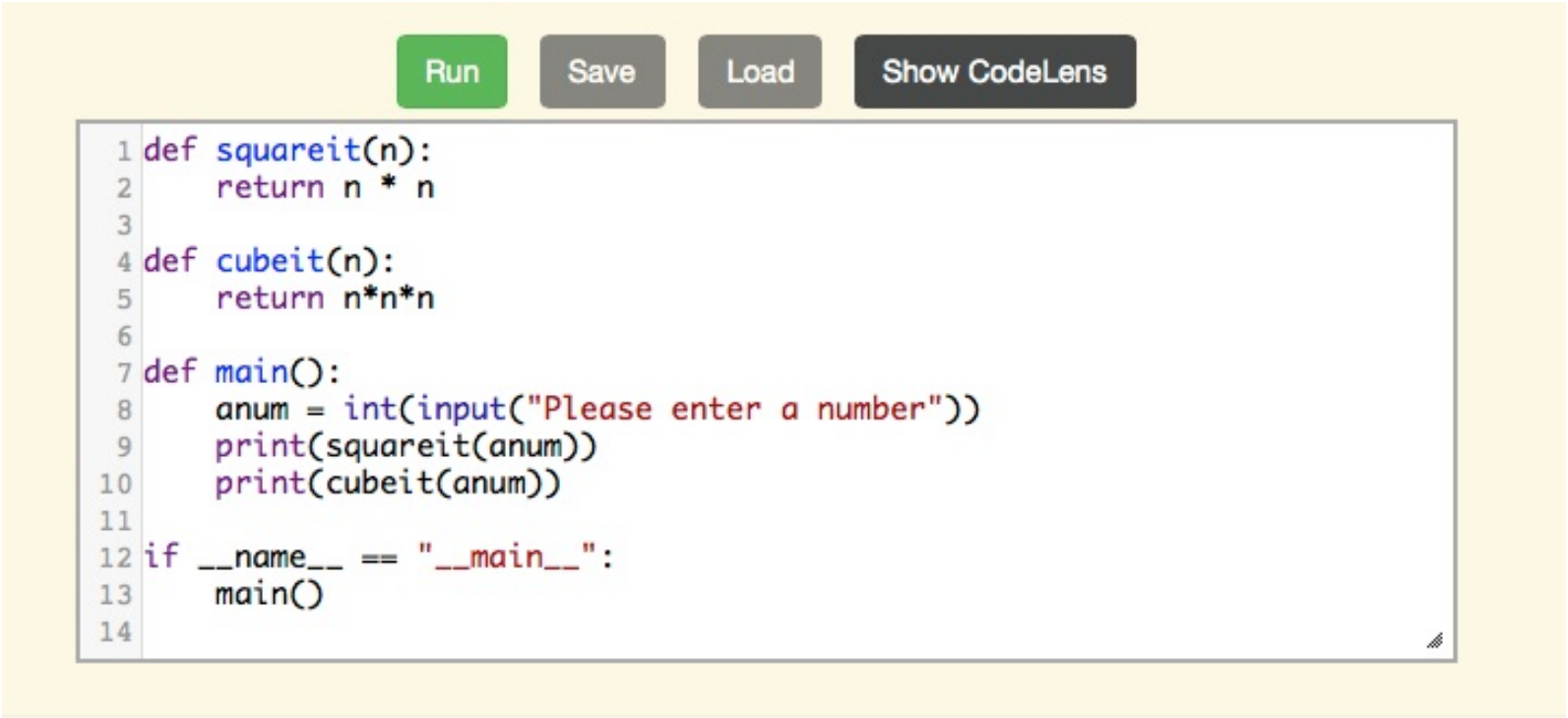
- And if-statements can be nested inside loops:

```
for i in range(5):
    for j in range(5):
        if i+j % 2 == 0:
            print "+",
        else:
            print "-",
    print
```

- As long as you indent correctly, you can nest loops and decisions.
- *Think CS* examples

Conditional Execution

To make a file that runs as program, but whose functions we can use separately, we use conditional execution:



Think CS example

Group Work: Using Decisions in Programs

In pairs/triples, work out (hints & answers at *Think CS* Selection Exercises):

- #1: What do these expressions evaluate to:
  ```
  3 == 3
  3 != 3
  3 > 4
  not (3 < 5)
  ```
- #3:

3. Write a function which is given an exam mark, and it returns a string — the grade for that mark — according to this scheme:

Mark	Grade
>= 90	A
[80-90)	B
[70-80)	C
[60-70)	D
< 60	F

The square and round brackets denote closed and open intervals. A closed interval includes the number, and open interval excludes it. So 79.99999 gets grade C , but 80 gets grade B.

Test your function by printing the mark and the grade for a number of different marks.

- #2: Give the logical opposites of these conditions. You are not allowed to use the `not` operator:
  ```
  a > b
  a >= b
  a >= 18 an day == 3
  a >= 10 or day != 4
  ```
- #13:

13. Implement the calculator for the date of Easter.

The following algorithm computes the date for Easter Sunday for any year between 1900 to 2099.

Ask the user to enter a year. Compute the following:

1. a = year % 19
2. b = year % 4
3. c = year % 7
4. d = (19 * a + 24) % 30
5. e = (2 * b + 4 * c + 6 * d + 5) % 7
6. dateofeaster = 22 + d + e

Special note: The algorithm can give a date in April. Also, if the year is one of four special years (1954, 1981, 2049, or 2076) then subtract 7 from the date.

Your program should print an error message if the user provides a date that is out of range.

Recap

- More on loops on Wednesday, 1pm.

Recap

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

Recap

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