Introduction to Python

Part 2: Loops and functions

COMP 089H Fall 2015

Intro to Python: part 2

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O

Type: list

A `list` is a collection of objects.

These can be anything, including other `lists` (these are *nested*).

You can put a `list` in a variable.

```
>>> myList = [“apple”, 42, 3.14]
>>> myList
[‘apple’, 42, 3.14]

>>> otherList = [7, ‘a’, myList]
>>> otherList
[7, ‘a’, [‘apple’, 42, 3.14]]
```

Type: list

A `list` is a collection of objects.

To access or modify an element of a `list`, use [ ].

In Python, 0 is the first element's index.

```
>>> myList = [“apple”, 42, 3.14]
>>> myList
[‘apple’, 42, 3.14]

>>> myList[0]
‘apple’

>>> myList[0] = “banana”
>>> myList
[‘banana’, 42, 3.14]
```

Type: list

A `list` is a collection of objects.

To access or modify an element of a `list`, use [ ].

In Python, 0 is the first element's index.

You can access the last with -1.

```
>>> myList = [“apple”, 42, 3.14]
>>> myList
[‘apple’, 42, 3.14]

>>> myList[0] = “banana”
>>> myList
[‘banana’, 42, 3.14]

>>> myList[-1] = “hi there”
>>> myList
[‘banana’, 42, ‘hi there’]
```

Type: list

A `list` is a collection of objects.

To access or modify an element of a `list`, use [ ].

To access a range of elements, use a colon. If you include numbers, those are the start (inclusive) and end (exclusive).

```
>>> myList = [“apple”, 42, 3.14]
>>> myList
[‘apple’, 42, 3.14]

>>> myList[:1] # start at beg.
[‘apple’]

>>> myList[1:] # 1 to end
[42, 3.14]
```

Type: list

A `list` is a collection of objects.

You can add two `lists` using +.

```
>>> myList = [“apple”, 42, 3.14]
>>> otherList = [7, “banana”]
>>> myList + otherList
[‘apple’, 42, 3.14, 7, ‘banana’]
```

Type: list

A `list` is a collection of objects.

```
>>> myList = [“apple”, 42, 3.14]
>>> otherList = [7, “banana”]
>>> myList + otherList
[‘apple’, 42, 3.14, 7, ‘banana’]
```

You can add two `lists` using +.

To add a single element at the end, use the *list method* `append`.

```
>>> myList
[‘apple’, 42, 3.14]
>>> myList.append(“peach”)
>>> myList
[‘apple’, 42, 3.14, ‘peach’]
```

Intro to Python: part 2

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops (for-each, for-index)
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O

Loops: motivation

Say you want to echo each letter of a word…

```
word = raw_input("Please enter a word: ")

print word[0]
print word[1]
print word[2]
print word[3]
```

Loops: motivation

Say you want to echo each letter of a word…

```
word = raw_input("Please enter a word: ")

print word[0]
print word[1]
print word[2]
print word[3]
```

If you try this, you could have errors if the user doesn't enter a word long enough, or incorrect behavior or if it's too long.

```
>>> ================================ RESTART ================================
>>>
Please enter a word: cat
c
a
t

Traceback (most recent call last):
  File "D:\My Documents\UNC\COMP 089H\printing_letters.py", line 9, in <module>
    print word[3]
IndexError: string index out of range
>>>
```

Loops: motivation

Say you want to echo each letter of a word…

To fix this, you could check the length of the word using the `len` function.

This is really long, and *hard-coded*.

```
word = raw_input("Please enter a word: ")

if len(word) == 1:
    print word[0]
elif len(word) == 2:
    print word[0]
    print word[1]
elif len(word) == 3:
    print word[0]
    print word[1]
    print word[2]
elif len(word) == 4:
    print word[0]
    print word[1]
    print word[2]
    print word[3]
```

for loops: for-each

The `for` keyword lets us loop over *each element* in an *iterable*.

```
>>> for letter in “hello!”:
        print letter


h
e
l
l
o
!
```

for loops: for-each

The `for` keyword lets us loop over each element in an *iterable*.

The variable between `for` and `in` is named by you. It is assigned to each element (letter in a string, value in a list) one after the other.

```
>>> for letter in “hello!”:
        print letter


h
e
l
l
o
!
```

for loops: for-each

The `for` keyword lets us loop over each element in an *iterable*.

The variable between `for` and `in` is named by you. It is assigned to each element (letter in a string, value in a list) one after the other.

You can also do a `for` loop over a variable (as long as it's *iterable*).

```
>>> for letter in "hello!":
        print letter


h
e
l
l
o
!
```

```
>>> myList = [1, 4, 9]
>>> for val in myList:
        print val


1
4
9
```

for loops: for-each

The `for` keyword lets us loop over each element in an *iterable*.

If the value before the colon is not an *iterable*, Python throws an error.

So far, we've seen the *iterables* `str` and `list`.

```
>>> for val in 4:
        print val


Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    for val in 4:
TypeError: 'int' object is not iterable
>>>
```

for loops: for-index

All `for` loops in Python are for-each loops, meaning the variable gets the value in the iterable.

Sometimes you want its position, too.

Python provides the function `range`.

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2,5)
[2, 3, 4]
>>> range(1,10,1)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(
range([start,] stop[, step]) -> list of integers
```

for loops: for-index

You can combine `range` with `len` to iterate over the indices of an iterable, and then use `[ ]` to access each element.

(This is similar to how you might use for loops in some other languages.)

```
>>> myList = [“apple”, “banana”, “cantaloupe”]
>>> for i in range(len(myList)):
        print “The fruit at index”, i, “is”, myList[i]

The fruit at index 0 is apple
The fruit at index 1 is banana
The fruit at index 2 is cantaloupe
```

for loops: for-index

You can combine `range` with `len` to iterate over the indices of an iterable, and then use `[ ]` to access each element.

Keep in mind, the for loop is still for-each, but now it's over the list formed by range.

```
>>> myList = ["apple", "banana", "cantaloupe"]
>>> for i in range(len(myList)):
        print "The fruit at index", i, "is", myList[i]

The fruit at index 0 is apple
The fruit at index 1 is banana
The fruit at index 2 is cantaloupe

# Compare
>>> myList = ["apple", "banana", "cantaloupe"]
>>> length = len(myList)
>>> length
3
>>> indexRange = range(length)
>>> indexRange
[0, 1, 2]
>>> for i in indexRange: # for-each over indexRange
        print "The fruit at index", i, "is", myList[i]

The fruit at index 0 is apple
The fruit at index 1 is banana
The fruit at index 2 is cantaloupe
```

while loops

Sometimes you want to repeat *until* something happens.

For example, you could echo a user until they type a specific stop-word.

```
>>> ans = ''
>>> while ans != '.':
        ans = raw_input("Please enter a word, or '.' to quit: ")
        print "You said:", ans


Please enter a word, or '.' To quit: apple
You said: apple
Please enter a word, or '.' To quit: banana
You said: banana
Please enter a word, or '.' To quit: .
You said: .
>>>
```

while loops

Be careful about infinite loops!

Make sure you change whatever value the while condition checks.

Type Ctrl-C in IDLE to cancel a command if this happens.

```
>>> num = 1
>>> while num <= 10: # I will never stop :o
        print num * num

1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

while loops

Be careful about infinite loops!

Make sure you change whatever value the while condition checks.

Type Ctrl-C in IDLE to cancel a command if this happens.

```
>>> num = 1
>>> while num <= 10:
        print num * num
        num = num + 1 # much better


1
4
9
16
25
36
49
64
81
100
```

Intro to Python: part 2

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops (for-each, for-index)
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O

Functions

We've already seen a handful of built-in functions:

- `int`
- `str`
- `raw_input`
- `range`
- `len`

We've also seen the `list` method `append`.

Functions

You can create your own with the `def` keyword.

By convention, function names in Python start with lowercase letters.

IDLE will turn your function's name blue in the *definition*.

```
>>> def myAddFunction(a, b):
        print a + b

>>> myAddFunction(2, 6)
8
```

```
>>> def sayHi(): # don't need params
        print "Hi!"

>>> sayHi()
Hi!
```

Functions

If your function only prints, you can't call it and expect to use its value in other expressions.

```
>>> def myAddFunction(a, b):
        print a + b


>>> myAddFunction(1,2) * myAddFunction(3,4) # expect: 3 * 7 = 21
3
7

Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    myAddFunction(1,2) * myAddFunction(3,4) # expect: 3 * 7 = 21
TypeError: unsupported operand type(s) for *: 'NoneType' and 'NoneType'
>>>
```

Functions

If you want to use its result, you have to *return* that result.

```
TypeError: unsupported operand type(s) for *: 'NoneType' a
>>>
>>> def myBetterAddFunction(a, b):
        return a + b

>>> myBetterAddFunction(1,2) * myBetterAddFunction(3,4)
21
>>>
```

```
>>>
>>> firstNum = myBetterAddFunction(1,2)
>>> secondNum = myBetterAddFunction(3,4)
>>> firstNum * secondNum
21
>>>
```

Intro to Python: part 2

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops (for-each, for-index)
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O

Built-in modules

Python has a variety of built-in modules, which you can use via the `import` keyword.

For example, the `math` module provides functions like `sqrt`.

To call them, provide the module name: "`math.methodName`".

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.sin(math.pi) # note: e-16 is about 0
1.2246467991473532e-16
```

External packages

You can also download useful Python packages, such as:

- TkInter: graphical user interfaces
- Pygame: games
- NumPy/SciPy/Matplotlib: scientific computing, plotting
- Python Imaging Library (PIL): image manipulation

https://wiki.python.org/moin/TkInter

http://www.pygame.org/hifi.html

http://www.numpy.org/

http://www.scipy.org/

http://pythonware.com/products/pil/

Intro to Python: part 2

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops (for-each, for-index)
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O

File Input/Output

You can read and write to a file.

```
>>> f = open("myText.txt", 'r')
>>> f.close() # close when done!
```

Use `open` to get a variable for the file – provide the name and your access mode:

- 'r': read
- 'w': write
- 'a': append (add to the end)

File Input/Output

You can read and write to a file.

A file is an *iterable*, so you can use it in a `for` loop to process the lines.

```
>>> f = open("myText.txt", 'r')
>>> for line in f:
        print line
// My favorite foods

popcorn

apples, green grapes
>>> f.close() # close when done!
```

File Input/Output

You can read and write to a file.

There are empty lines because text files have *new line characters*, like "\n" or "\r\n".

You can remove leading/trailing whitespace ('\n', '\r\n', '\t', ' ') with the string method `strip`.

```
>>> f = open("myText.txt", 'r')
>>> for line in f:
        print line.strip()
// My favorite foods
popcorn
apples, green grapes
>>> f.close() # close when done!
```

File Input/Output

You can read and write to a file.

You can use `split` to divide a `str` into a `list` of sub-strings, by a *separator* of your choice.

```
>>> f = open("myText.txt", 'r')
>>> for line in f:
        s = line.strip()
        vals = s.split(',')
        print vals
['// My favorite foods']
['popcorn']
['apples', ' green grapes']
>>> f.close() # close when done!
```

File Input/Output

You can read and write to a file.

You can use `split` to divide a `str` into a `list` of sub-strings, by a *separator* of your choice.

You might need to use `strip` again to remove new whitespace.

```
>>> f = open("myText.txt", 'r')
>>> for line in f:
        s = line.strip()
        vals = s.split(',')
        for val in vals:
            print val.strip()
// My favorite foods
popcorn
apples
green grapes
>>> f.close() # close when done!
```

File Input/Output

You can read and write to a file.

You might want to ignore certain lines, such as comments that start with # or //.

The `str` method `startswith` returns a `bool`.

```
>>> f = open("myText.txt", 'r')
>>> for line in f:
        s = line.strip()
        if not s.startswith("//"):
            vals = s.split(',')
            for val in vals:
                print val.strip()
popcorn
apples
green grapes
>>> f.close() # close when done!
```

File Input/Output

You can read and write to a file.

To write to a file, open it with `'w'` or `'a'`, and use the `write` method.

Don't forget to add new-line characters if you want separate lines!

```
>>> myList = ['a', 'b', 'c']
>>> f = open("newfile.txt", 'w')
>>> for letter in myList:
        f.write(letter + '\n')
>>> f.close() # close when done!
```

```
# Newfile.txt contains:
a
b
c
```

Today we covered:

- Type: list
  - Creating
  - Accessing, modifying elements
  - Method: append
- Loops
  - for loops (for-each, for-index)
  - while loops
- Functions
  - Defining
  - Print vs. return
- More tools
  - Built-in modules
  - External packages
- File I/O