



ECE 364 Software Engineering Tools Lab

Lecture 3 Python: Introduction



1

Lecture Summary

- Introduction to Python
- Common Data Types
- If Statements
- For and While Loops
- Basic I/O



2

What is Python?

- Python is a flexible programming language
 - Procedural like C
 - Object-oriented like C++, Java and C#
 - Functional like LISP, Haskell, Scheme, Scala
- Python is a high-level language
 - Has a rich set of high-level data types and functions (standard library)
 - Runs on many platforms and operating systems
 - Windows, Linux/Unix, Mac OS and much more...



3

What is Python?

- Python is designed to be simplistic
 - Very simple and consistent syntax
 - Easy to read and write
 - Interactive like a shell



4

What is Python?

- Most Python implementations have an interactive mode
 - Some implementations provide more interactive features than others
 - Examples: IDLE, IPython



5

Syntax and Features

- No braces { }
- Indentation is used to indicate code blocks
- No semicolon at the end of a statement
 - Unless needed to separate multiple statements on one line
- No \$ preceding variable names



6

Making a Python Script

- Put the Shebang: `#!` on the first line

```
#!/usr/bin/env python3.4
```
- This line invokes the `env` program to get the path to python
- Comments are indicated using a `#` mark.



7

The Traceback

- When Python crashes, you will see a message like
Traceback (most recent call last):
`File "./tables.py", line 12, in <module>`
`x_start = float(sys.argv[3])`
`ValueError: invalid literal for float(): c`
- Read it carefully, starting from the bottom and working upwards.
 - It is usually very accurate in pinpointing errors
 - It can save you a lot of time debugging
 - Debugging Python is easy compared to Bash



8

Interactive Mode

- Interactive Python:
 - Useful for testing out things you're not sure about
 - Useful for prototyping algorithms
 - Invoke at the command line with:
`python`
`ipython`
`idle` (graphical version)
- Comes with a built in help function:
- ```
>>> help(list)
>>> import sys
>>> help(sys)
```



9

## Data Types and Dynamic Typing

- Unlike C, variables are not declared with types
  - The C language uses "static typing"
  - All variables must be given an explicit type during before you can compile your program
- Semantics are defined by the set of functions and properties of the object
  - No need to define variable, nor define their type.
  - "If it looks like a duck and quacks like a duck, it must be a duck"<sup>1</sup>
- Check the data-type of any variable using the `type()` function

1. Source: <http://docs.python.org/glossary.html#term-duck-typing>



10

## Built-in Constants

- Python defines several constants
  - `True` – true value
  - `False` – false value
  - `None` – indicates the absence of value, like NULL
- Do NOT mistake with strings:
  - `'True'`, `'False'`, `'None'`



11

## Built-in Numeric Data Types

- `int` – Unlimited magnitude integer
  - `1234567890123456789012345678901234567`
  - Size only limited by available memory (and your time)
- `float` – Floating point number
  - 64-bit double precision numbers
- `complex` – Complex numbers
  - Represented as two floating point values
  - `Z = 1.2 + 7.5j`



12

## Truthiness

- Python will evaluate any object for truth in the context of a conditional or Boolean operation
- The following values will *evaluate* to **False**
  - False**
  - None**
  - 0, 0.0, 0j**
  - (), [], ''**



13

## Truthiness (2)

- Anything that does not evaluate to **False** evaluates to **True**
- Just because it evaluates to **False** does not mean it is **False**
  - An empty tuple **()** is not an empty list **[]**
  - None** evaluates to **False** but **False != None**
  - But **0.0** is **0**



14

## Comparison Operators

| Operator          | Function                                              |
|-------------------|-------------------------------------------------------|
| <b>A == B</b>     | <b>True</b> if A is <b>equal</b> to B                 |
| <b>A != B</b>     | <b>True</b> if A is <b>not equal</b> to B             |
| <b>A &lt; B</b>   | <b>True</b> if A is <b>less than</b> B                |
| <b>A &lt;= B</b>  | <b>True</b> if A is <b>less than or equal to</b> B    |
| <b>A &gt; B</b>   | <b>True</b> if A is <b>greater than</b> B             |
| <b>A &gt;= B</b>  | <b>True</b> if A is <b>greater than or equal to</b> B |
| <b>A in B</b>     | <b>True</b> if A is an <b>element in</b> B            |
| <b>A not in B</b> | <b>True</b> if A is <b>not an element in</b> B        |
| <b>A is B</b>     | <b>True</b> if A has the <b>object identity</b> B     |

Note: the **in** operator only works with collections



15

## Boolean Operators

| Operator       | Function                                    |
|----------------|---------------------------------------------|
| <b>A and B</b> | <b>True</b> if <b>A and B are both True</b> |
| <b>A or B</b>  | <b>True</b> if <b>either A or B is True</b> |
| <b>not A</b>   | <b>True</b> if <b>A is False</b>            |

Do NOT confuse with bitwise operators:

**&, |, and ~**



16

## Operator Precedence

- Precedence is listed from **highest to lowest**
  - \*, /, %**
  - +, -**
  - <, <=, >, >=, !=, ==**
  - in, not in**
  - not**
  - and**
  - or**
- Parenthesis are used to change the evaluation order, or improve readability.



17

## Strings

- Single quotes in pairs allow **"** without escaping
- Double quotes in pairs allow **'** without escaping
- Triple quotes (single or double) in pairs allow either single or double quotes and newlines with escaping
- There are no other differences in quote types.

```
X = "This isn't \"funny\" -- \\' not needed"
X = 'This isn\'t "funny" -- \\\' needed'
Y = "Don't quote \"me\" -- \\' not needed"
Y = 'Don\'t quote "me" -- \\' not needed'
Z = ""Don't quote "me""
Z = '''Don't quote "me'''
```



18

### Strings (1)

- `str` – A string of characters
  - `My_string = "Hello World!"`
  - `Name = "Goldfarb"`
- Use the `len()` function to get the length of any string
  - `len(My_String)`           # 12
  - `len(Name)`               # 8



19

### Strings (2)

- Strings can be easily concatenated using the addition (+) operator
  - `Var1 = "Hello" + " World"`
  - `Var2 = Var1 + ", how is the weather?"`



20

### Strings (3)

- `StrVar.rstrip()`  
returns a **new copy** of the string with whitespace removed from the right side.
- `StrVar.lstrip()`  
returns a **new copy** of the string with whitespace removed from the left side.
- `StrVar.strip()`  
returns a **new copy** of the string with whitespace removed from both sides.



21

### Strings (4)

- `StrVar.split()`  
splits `StrVar` into a list on **whitespace**  
  
*Q: What is the difference between:  
`StrVar.split()` and `StrVar.split(" ")` ?*
- `StrVar.split(delim)`  
splits `StrVar` into a list on the string `delim`
- `StrVar.split(delim, n)`  
splits `StrVar` into a list on the first `n` occurrences of the string `delim`



22

### Strings (5)

```
Data = " mgoldfar,ee364a01,10.6 "

Clean off any extra whitespace
Data = Data.strip()

Split into list of strings
Cols = Data.split(",")

Cols[0] is 'mgoldfar'
Cols[1] is 'ee364a01'
Cols[2] is the string '10.6'
```



23

### Strings (6)

- `delim.join(StrList)`  
returns a string with each string in `StrList` separated by the string `delim`  
  
`Sep = "?"`  
`Items = ["Baz", "Foo", "Bar"]`  
`ItemsStr = Sep.join(Items)`  
  
# `ItemsStr` is "Baz?Foo?Bar"
- You can inline the separator string also!  
`ItemsStr = ",".join(Items)`  
  
# `ItemsStr` is "Baz, Foo, Bar"



24

## Lists

- **list** is a built-in Python data type
  - Much more powerful than plain old arrays
  - Can also be used to implement **stacks** and **queues**
- Lists are containers of things (objects)
  - Items need not be of the same data type

```
A = [1, 2, 3, 4]
B = [1, "Big Deal", [1, 2], 6.7]
```



25

## Lists (2)

- Lists are **mutable**, elements can be reassigned:  

```
A = [1, 2, 3]
A[0] = "First"
```
- Use the **len(X)** function to return the length of a list  

```
len(A) # Returns 3
```
- Lists are **not** sparse – an index must exist  

```
A[9] = "foo" # Illegal - causes a runtime error
```



26

## Indexing

Negative indices are allowed in Python

```
X = ["1st", "2nd", "3rd"]
X[0] = X[-3] = "1st"
X[1] = X[-2] = "2nd"
X[2] = X[-1] = "3rd"
```

- 0 is the index of the **leftmost** item
- -1 is the index of the **rightmost** item



27

## Slicing

- Slicing is a way to extract multiple elements from lists, tuples and strings.

**A[M:N]** A slice of elements starting from index **M** and ending at index **N-1**

**A[M:N:S]** A slice of elements starting from index **M** and ending at index **N-1**, with a step **S**

**A[M:]** A slice of elements starting from index **M**

**A[:N]** A slice of elements starting from index **0** and ending at index **N-1**

**A[:]** A slice containing all elements of A



28

## Slicing (2)

- Many things in Python can be sliced.
  - List, tuples and strings just to name a few

```
A = [1, 2, 3, 4, 5]
B = "ECE 364 is only 1 credit hour."
```

```
A[2:4] is [3, 4] B[4:7] is '364'
```

```
A[:3] is [1, 2, 3]
```



29

## List Functions

- **list.append(x):** Add an item to the end of the list.  

```
l = [2, 5]
l.append(13) # l = [2, 5, 13]
```
- **list.extend(L):** Extend the list by appending all the items in the given list.  

```
l = [2, 5]
g = [7, -6]
l.extend(g) # l = [2, 5, 7, -6]
```
- Another way to extend a list is to use the addition operator (+)
  - Note: You must use a list on both sides of the operator!

```
T = ["Mike", "Greg"]
T = T + ["Sudhir"] # Or T += ["Sudhir"]
T is ['Mike', 'Greg', 'Sudhir']
```



30

## List Functions (2)

- `list.insert(i, x)`: Insert an item at a given position.
- `list.remove(x)`: Remove the first item from the list whose value is x. It is an error if there is no such item.
- `list.clear()`: Remove all items from the list. Equivalent to `del a[:]`.
- `list.index(x)`: Return the index in the list of the first item whose value is x. It is an error if there is no such item.
- You can use the `in` operator to test for membership.  

```
m = [3, 2, 6, 11, 2, 15, 77]
9 in m # Result in False
11 in m # Result in True
```
- `list.count(x)`: Return the number of times x appears in the list.



31

## List Functions (3)

- `list.sort()`: Sort the items of the list in place.
- `list.reverse()`: Reverse the elements of the list in place.
- `list.copy()`: Return a copy of the list. Equivalent to `b=a[:]`.  
Note: `copy` creates a shallow copy.  
# If you do `c = l` instead of using `copy()`, then `c` & `l` both point to ['IN', 'MI', 'OH', 'IL', 'CA'].  
# If you change `l`, `c` will also change.
- `del list[i:j]`: Removes the items `i` to `j-1` from the list. (You can also use `del` with/without indexing.)



32

## File I/O

- The preferred method to open files is using the `with` keyword.
- The `with` keyword is a shorthand for a lot of work in the background to ensure resources are claimed by the system when done.
- Can be used for both reading and writing.



33

## File I/O (2)

Example 1: Reading the file as a list of lines.

```
This is called a "with-block"
The "myFile" below is called the file alias.
'lines' is a list of strings.
```

```
with open('textFile.txt', 'r') as myFile:
 lines = myFile.readlines()
```

Example 2: Reading the file as a single string.

```
'content' is a single string of the whole file.
```

```
with open('textFile.txt', 'r') as myFile:
 content = myFile.read()
```



34

## File I/O (3)

Example 3: Writing a list of strings.

```
The list of strings must already contain the "\n"
```

```
with open('textFile.txt', 'w') as myFile:
 myFile.writelines(lines)
```

Example 4: Writing a single string.

```
Construct the string before you write it.
```

```
with open('textFile.txt', 'w') as myFile:
 myFile.write(content)
```



35

## Type Conversion

- There is no casting from one type to another in Python
  - New values can be created from other values of different types
  - If a new value can not be created python will raise an exception

`int(x)` Attempts to create a new integer from x

`float(x)` Attempts to create a new float from x

`str(x)` Attempts to create a new string from x

`complex (re, im)` Attempts to create a complex number.



36

## Type Conversion (2)

`str(Var)`  
Create a new string from `Var`.

`int(StrVar)`  
Create a new integer from `StrVar`.

`int(StrVar, base)`  
Create a new integer from `StrVar` in base `base`.

`float(StrVar)`  
Create a new floating point number from `StrVar`.



37

## if Statement

- Behaves just like any other if statement you have encountered
- Colons are **required** at the end of each conditional expression

```
if <condition1>:
 statements
elif <condition2>:
 statements
else:
 statements
```

- Note: Python does not have a `switch` statement.



38

## for Loops

- Python supports only one style of for loop:

```
for <variable> in <sequence>:
 statements
```

- Example:

```
Sum = 0
for Item in [2, 4, -2, 5]:
 Sum = Sum + Item
print(Sum)
```

- Loop execution can be modified with **break** and **continue**



39

## while Loops

```
while <condition>:
 statements when condition is true
```

```
j = 10
while j > 0:
 print(j)
 j -= 1
```

- Avoid using **while** loops in place of **for** loops, as it is harder to debug.



40

## range Function

- Range can generate a lists of integers

`range(N)` Generates a list of integers between 0 and `N-1`  
`range(6)` → [0, 1, 2, 3, 4, 5]

`range(M, N)` Generates a list of integers between `M` and `N-1`  
`range(5, 10)` → [5, 6, 7, 8, 9]

`range(M,N,S)` Generates a list of integers between `M` and `N-1` with a stride of `S`.  
`range(10,50,10)` → [10, 20, 30, 40]



41

## range Function (2)

- Try your absolute best **NOT** to use the index, and operate on the elements directly.
  - This is the *Pythonic* Style. Works with all collection types:

```
for item in somelist:
 # Do Something with the item.
```

- Works only with lists & tuples. (NOT Recommended):

```
for index in range(10):
 # Get the item from the list.
 # Do Something.
```



42

## Printing

- Use `print(someString)` to write data to `stdout`

```
print("Hello World!")
```

- New way to control printing is by using `format()`
  - Old Style (with `%`) can still be used, but not recommended.
- Examples:
  - Print one or more elements in order.

```
>>> print('I am "{}" and I am "{}".format('Smart', 'Happy'))
I am "Smart" and I am "Happy".
```
  - Print one or more elements out of order.

```
>>> print('I am "{}" and I am "{}".format('Smart', 'Happy'))
I am "Happy" and I am "Smart".
```



43

## Printing (2)

- Examples:
  - Printing with keywords.

```
>>> print('Today is "{}" and it is "{}"!'.format(weather='Sunny', day='Monday'))
Today is "Monday" and it is "Sunny".
```
  - Print one or more elements out of order.

```
>>> print('I am "{}" and I am "{}".format('Smart', 'Happy'))
I am "Happy" and I am "Smart".
```
  - Print an integer with leading 0s.

```
>>> print("{}:05d".format(23))
'00023'
```
- Check string formatting specs for more details:  
<https://docs.python.org/3/library/string.html#formatstrings>



44

## Python Modules

- Python has a few built-in functions
- Many more are found in `modules`
  - A module is like a header file, it provides access to new functions
  - Common modules: `sys`, `string`, `os`, and `math`
- When you want to use a function from a module, you must first import that module

```
import sys
import os, math
from enum import Enum
from pprint import pprint as pp
```



45

## Python Modules (2)

- A common problem is not importing a module when it is used
  - Python will raise an error if you forget to import something
- Example: (Solution: add `import sys`)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
NameError: name 'sys' is not defined



46