Fairfield University
DOLAN
School of Business

Python for Analytics

Python Fundamentals
RSI Chapters 1 and 2

Learning Objectives

- **Theory:** You should be able to explain ...
  - General programming terms like *source code*, *interpreter*, *compiler*, *object code*, *comment*, *data type*, etc.
  - Basic Python syntax and structure, including statements, variables, expressions, operators, and functions
  - The different types of errors that require debugging
- **Skills:** You should know how to ...
  - Run Python statements in the command line interpreter
  - Use variables to store, retrieve, and update values
  - Evaluate arithmetic and string expressions

Python Language Origins

- Python is a high-level *scripting* language, originally intended for short programs that run from the command line.
- Timeline:
  - 1980's: early development by Guido Van Rossum
  - 1994: Python 1.0, the first complete release
  - 2000: Python 2.0 added advanced data types and core object-orientation ('*everything* is an object')
  - 2008: Python 3.0 broke backwards compatibility to streamline and unify language syntax and libraries

Python 2 vs Python 3

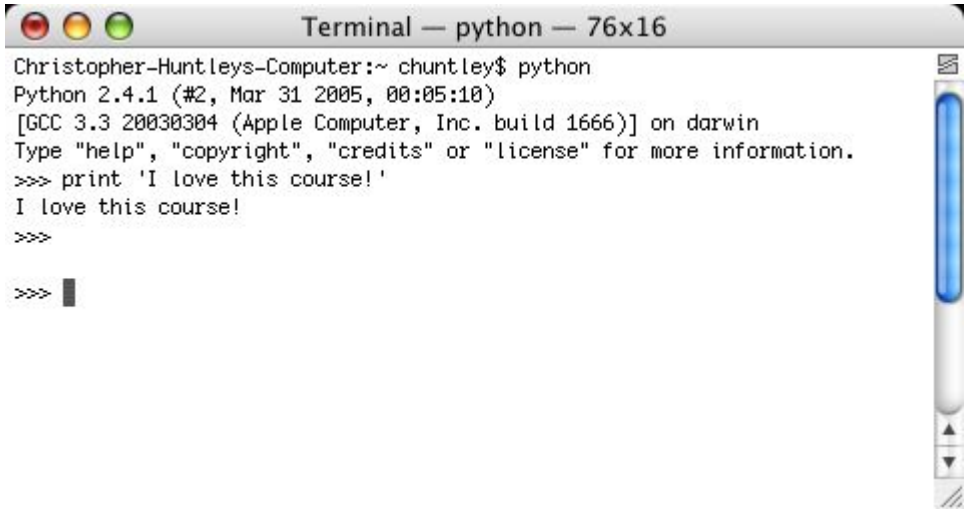
- Plenty of Python libraries (reusable programs) in use today were originally written before 2008
  - While most libraries have updated to Python 3, some remain stuck in Python 2
- **We will be learning Python 3** in this class
  - However, you may be asked to use Python 2 in some of your future analytics classes and on the job
  - Fortunately, the switch is pretty easy once you know what you are doing

Source Code, Compilers, and Interpreters

- Python is a high-level language like C, Java, or C#
  - Written and read by humans as **source code**
  - Has to be converted into low-level **object code** (not human-readable) that the computer uses natively
- Two ways to convert from source code (programs) to object code (machine code):
  - **Compilers** (used by C, Java, C#, etc.) *convert it all at once*, requiring all source code to be *written in advance*
  - **Interpreters** *convert the code one line at a time*, allowing the source code to be written *interactively*

Python Interpreter by Example

- Most Python programs are executed via interpreter
- Anaconda supports a command prompt and Jupyter Notebooks



Fun fact: This example is from Dr. Huntley's first Python class in 2005

Interpreter Notes

- Executes a program one **statement** at a time
- We entered a Python 2.4 statement

```
print ‘I love this course!’
```

  In Python 3 it would be

```
print(‘I love this course!’)
```

- The interpreter **executed** the statement
- If the interpreter cannot execute the statement, then it returns an **error message**

Try it yourself!

1. Open the Python interpreter from the command line:
   - Mac: Open Terminal.app from Utilities folder
   - Windows: Open the Command Prompt (Google it)
2. Enter the statement

   ```
   print('I love this course')
   ```
3. Watch as the interpreter executes the statement
4. Enter the statement

   ```
   print 'I love this course'
   ```
5. Read the error message about missing parentheses

Debugging / Error Types

These are listed in increasing difficulty. Logic errors can be especially hard to diagnose and correct.

- **Syntax errors** in the source code are caught by the interpreter or compiler before trying to run it
  - Fix: Look for broken Python statements or typos
- **Runtime errors** happen when the computer tries to run a line of code
  - Fix: Read the error message, which explains what caused the interpreter to break
- Programs with **Semantic errors** (bad logic) run fine but do not produce the expected results
  - Fix: Study the source code for **incorrect logic**

Code Comments

```
# Compute the W2H ratio for one person
```

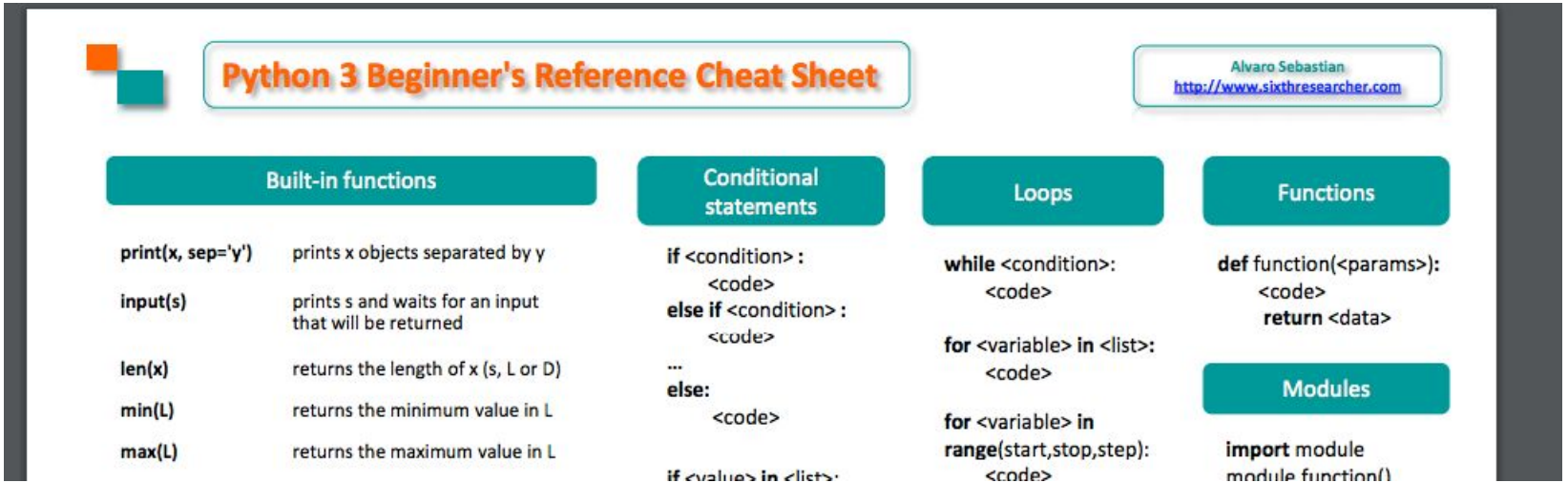
- Python statements are meant for the **computer** to **execute**
- Sometimes we want to include explanatory ***comments*** for **programmers** to **read**
  - Comments can be especially helpful with semantic errors
- Comments always start with the # character
  - Anything after the # is ignored by the interpreter

Notes on Syntax / Notation

- Python is very strict about what kinds of statements it can execute
- Every kind of statement has its own **syntax**, a pattern composed of **keywords**, *expressions*, and **punctuation characters**.
- The syntax of a `print` statement is
  - **`print`**`(`*`<string expression>`*`)`
- **Keywords** like **`print`**, **`if`**, **`or`**, and **`for`** are **reserved** and *cannot be used for variable names*

Python Cheat Sheet

Suggestion: Download the *Python 3 Beginner's Reference Cheat Sheet* from sixthresearcher.com



Values and Data Types

Every piece of data in Python has a **value** and a **data type:**

- `5` is a value with the data type `int` (integer)
- `5.0` is a value of type `float` (floating point number)
- `'Hello'` is a value of type `str` (string of characters)
- `['IS505',2017]` is a `list` (of values)

Try it out

Type the following lines in the Python Interpreter

- `type(5)`
- `type(5.0)`
- `type('5.0')`
- `type("Hello")`
- `type(['IS505',2017])`

For each you will get something like

`<class 'data type'>`

Notice anything about the value `'5.0'`? What happened?

Data Types Matter!

The data type of a value determines what kinds of things we can do with it.

- `'A'+'B'` → `'AB'`
- `1+2` → `3`
- `1+2.0` → `3.0`
- `1+'A'` → ***error message***

The last example is an example of ***type incompatibility***

Type Conversion

We can translate data from one data type to another using **type conversion functions**

- `float(5)` → `5.0`
- `int(5.0)` → `5`
- `str(5)` → `'5'`
- `int('5.0')` → **syntax error! Why?**

We can do more complex conversions like this ...

`int(float('5.0'))` → `5`

Variables

A **variable** is just a place that stores a value that *we can recall* later if needed:

- Can store **one data value at a time**
- Has a **unique name** (within scope) so you can access the data
- Can change the data value through **assignment**

We can set the value of a variable using the **assignment operator (=)**

```
<variable name> = <value>
```

Try it out

```
first_name = “Bob”
last_name = “Gibson”
print(“Hi ”+first_name+“ ”+last_name)
    ⟶ Hi Bob Gibson
first_name = “Dan”
print(“Hi ”+first_name+“ ”+last_name)
    ⟶ Hi Dan Gibson
```

Variable Names

- Should be easy to understand and type
  - Make the names **describe what is being stored** and **how it is being used**
  - **`last_name`** is much better than names like **`X`** or **`l`** or even **`ln`**
- Follow the Python style guide
  - Variable names are lower case and use underscores to separate words (e.g., last_name)
  - Don’t go too crazy on the length (1-15 chars)
  - And do not use keywords for names

Statements and Expressions

A **statement** is a Python instruction (line of code) that asks Python to do something:

- `X=2` is an assignment statement

An **expression** is a combination of **values**, **variables**, **operators**, and **functions** that can be ***evaluated*** to calculate a **value**

- Expressions ***always*** evaluate to a value
- `2+2` and `type(2+2)` are expressions

Functions (more about this next time)

A **function** is a named, reusable sequence of statements

- Like how a variable is a place to store a value for later

Functions *can* **return** values, just like expressions

- `str()` is a function that **returns** a string

Function calls always include () after the function name

- Any values listed inside the () are **input arguments**
- `str(1.0 + 3)` calls `str()` with the expression `1.0+3` as the argument and returns the equivalent string value

Digression: The `input()` Function

The `input()` function allows us to ask the user a question directly from the Python interpreter.

- Execution pauses until the user enters a value (and hits the return key) before returning a string value

Usually, we will want to capture the return value with a variable

- `age = input("How old are you?")`

We won't be needing the `input()` function much in this class. Jupyter Notebooks are way cooler than the command line.

Composite Function Calls

Function arguments can call other functions if needed.

Remember this?

```
int(float('5.0'))
```

The `float('5.0')` function call inside the parentheses is evaluated before passing the value into the `int()` function.

Operators and Operands

An operator is a computation (‘verb’) that can be used in an expression to calculate a value

- `+`, `-`,`*`, and `/` are arithmetic operators
- `(` `)` is a grouping operator
- An **operator** is like a function that has special built-in ‘shortcut’ syntax:
  - `2 + 3` is equivalent to function call like `add(2,3)`
  - The values being operated on (2 and 3 above) are called **operands**

Order of Operations: PEMDAS

Math expressions are evaluated just like in algebra class:

1. **P**arentheses
2. **E**xponents
3. **M**ultiplication
4. **D**ivision
5. **A**ddition
6. **S**ubtraction

Trick: When in doubt use parentheses to force the right order.

Classwork to do before leaving for home

- Complete chapters 1 and 2 of the RSI *How to Think Like a Computer Scientist* e-book.
  - Do not play the videos without headphones; they are redundant anyway
- If time permits, start in on your homework.
- Ask questions when you need help. Use this time to get help from the professor!

Homework to be completed before the next class

The following homework is **due before class on Saturday**:

- RSI Chapters 3,4,and 10.
- Data Camp “Python Basics” and “Python Lists” chapters
- Study for Quiz 1, which will cover **chapters 1-4** of the RSI book

Please email chuntley@fairfield.edu if you have any problems or questions.

Fairfield University
DOLAN
School of Business

Python for Analytics

Python Fundamentals
RSI Chapters 1 and 2