Fairfield University DOLAN School of Business

Python for Analytics

Debugging, Libraries, Objects, Collections, and Control Structures
RSI Chapters 3 and 4

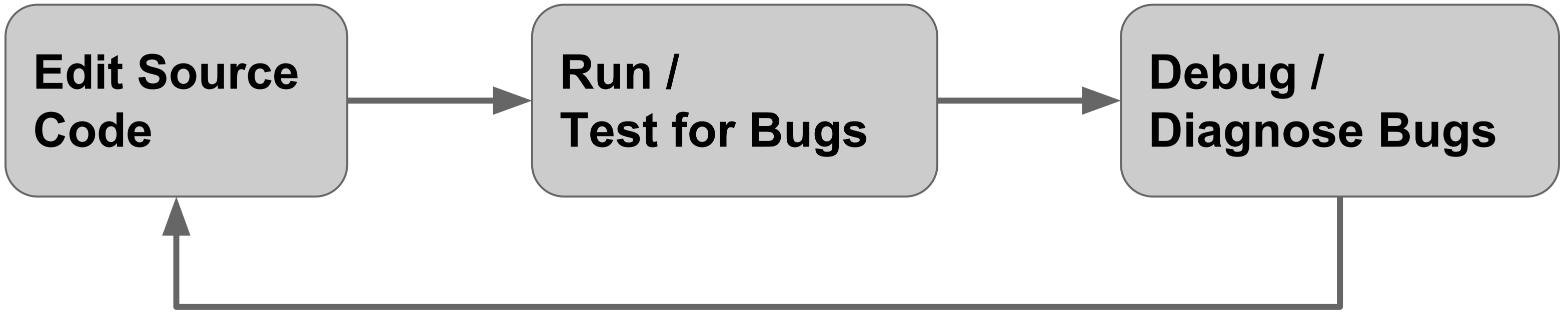
Learning Objectives

- **Theory:** You should be able to explain ...
  - The Edit-Run-Debug programming cycle
  - Common types of bugs to look for and how to find them
  - **Libraries, modules**, **classes, objects/instances**, **object state**, **methods**, **collections**, **control flow**, **iteration**, and **`for` loops**
- **Skills:** You should know how to ...
  - Interpret standard Python error messages
  - Call a method on an object
  - Use a `for` loop to iterate over a collection of objects

Debugging

Because Stuff Happens

Debugging is a Continual Process



This Edit-Run-Debug cycle goes on ***forever***.

- In 2014, a serious security bug nicknamed *Shellshock* was found in **bash**, a package installed on hundreds of millions of computers, including *every* Macintosh since 2001. **The bug had existed since 1989.** *The bug has been fixed, of course.*

Avoiding Bugs

1. **Know the Requirements.**

   Be smart but lazy. Do not solve the wrong problem and do not do more than you have to.

2. **Start Small.**

   Debug early and often, starting with your first few lines of source code.

3. **Keep Testing and Improving.**

   Even "perfect" code doesn't exist in a vacuum. Changes to other people's code can expose/induce bugs in your code!

Interpreting Error Messages

Often when things fail, Python will try to tell you what happened using standard error messages

Each error message has its own (obscure) way of reporting the problem.

Take the time to get to know the standard error messages and what they are telling you.

Most Common Errors

- **ParseError (55%)**
    - A statement cannot be understood by the interpreter. *Look for broken syntax. You may have to RTFM to get it right.*
- **TypeError (14%)**
    - An operation/function is being called on data of the wrong type. Check the data types and (again) *RTFM*.
- **NameError (11%)**
    - Usually, referring to a variable that does not exist. Check for typos in your variable names. Likely don’t have to *RTFM*.
- **ValueError (10%)**
    - Function/op arguments fail validation check. *RTFM*.

Diagnosis is an Active Process

Staring at your code to find and fix errors does not usually work. It just gives you eye strain.

Use a debugger (e.g., in Spyder IDE) or insert `print` statements (e.g., in Jupyter Notebook) to see what is going on.

Pro Tip: You can delete/comment out the print statements after you've fixed the bug. Or, use a logging tool to print debugging message to a separate file/console. The messages can be turned off and on as needed.

Fun with Turtle Graphics

Really, just an introduction to even more big concepts

Turtle Graphics is a Library

Turtle Graphics is a **library** for drawing line art on an area of the screen (called the canvas).

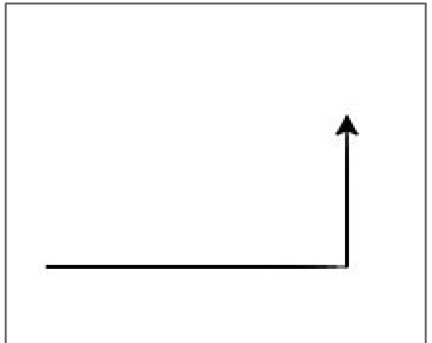
A **library** is a collection of reusable python code, organized into one or more **modules.**

Modules are files (whose names end in `.py`) with variables, callable functions and other reusable code.

We **`import`** modules to (re)use the code in our own programs.

A Simple Turtle Graphics Program

```
import turtle                # use the turtle library
wn = turtle.Screen()         # creates a graphics window
alex = turtle.Turtle()      # create a turtle named alex
alex.forward(150)           # tell alex to go forward 150 units
alex.left(90)               # tell alex to turn by 90 degrees
alex.forward(75)             # continues on (upward) ...
```



Classes, Objects, and Methods

- A **`class`** is a custom data type that defines ...
  - What kind of data is stored and how it is organized
  - What kinds of operations can be applied to the data

  Each data type in Python (even built-in ones like `int` or `float`) is actually a class.

- An **object** is an ***instance of*** a class (data type)
  - The functions/operations we can apply to the data are called **methods.**
  - The current value of an object's data is called its **state.**

  Each data value in Python is actually an object.

Dot Notation

To ***invoke*** (or call) a method of an object (literal value or variable reference), we tack on a dot (period) followed by the function call.

<object>.*<method call>*

More string methods are covered in RSI Chapter 9.

```
"ZXYW".lower()    # method call on a string literal
alpha="abcd"      # set a string variable
alpha.upper()     # method call on string variable
```

The Turtle Graphic Program (again)

```
import turtle                 # A module is just a container
wn = turtle.Screen()          #   object for classes, constants,
alex = turtle.Turtle()        #   variables, functions, etc.
alex.forward(150)             # We access the various parts of
alex.left(90)                 #   the module using dot
alex.forward(75)              #   notation, just like
                              #   any other object!
```

We'll come back to this later

Collections

Some data types can store collections of values:

- ***Tuples*** are used for read-only sequences of values, like an ordered pair (2,3) or triplet (4,2,8) in Math
  - Note: Python strings are just tuples of characters.
- ***Lists*** are like Tuples but can be altered after they have been created
- ***Sets*** never have duplicate values
- ***Dictionaries*** use a ***set of keys*** to index a ***set of values***. Each key acts like a variable with a value.

Collection Indexes

When we want to retrieve a ***member*** of a collection, we use an ***index*** to indicate which one we want.

```
my_tuple = tuple(“zxyw”)  # equivalent to (‘z’,‘x’,‘y’,‘w’)
my_list = [4,3,2,1]

second_letter = my_tuple[1]   # returns ‘x’
third_number = my_list[2]     # returns 2
```

Iteration with for Loops

You may want to RTFM about `for` loops and the `print` function.

When we apply the same instructions to each member of a collection, we are ***iterating*** over the collection.

```
values = [“x”,“y”,“z”]
print(values[0],values[1],values[2])     # the buggy way
for v in values:      # for loop with implicit enumerator
    print(v,end=“ ”)
for i in range(len(values)):      # for loop with range
    print(values[i],end=“ ”)
```

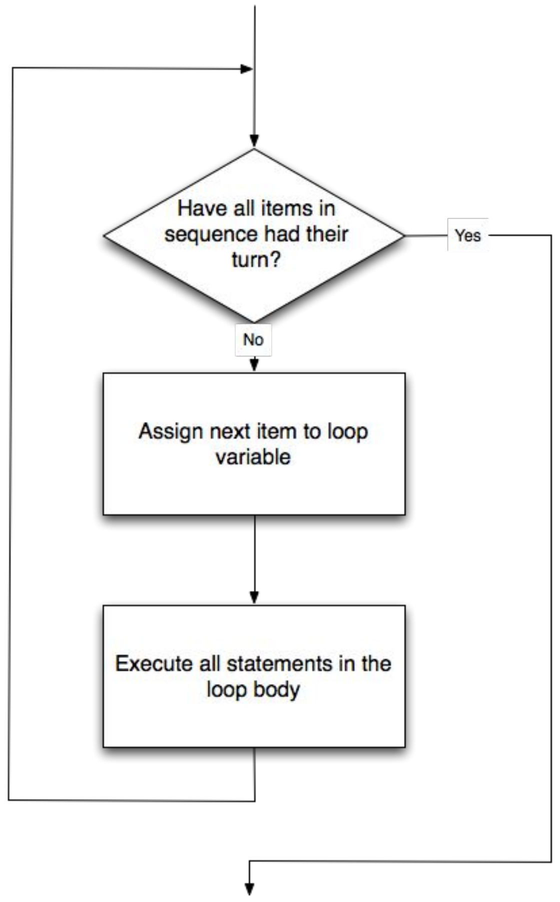
Control Flow / Structure

A loop is a kind of **control structure** that indicates the **flow of execution**.

Three basic kinds:

- Sequences (do *this* then do *that* ...)
- Loops (repeat this until done)
- Conditionals (if X then do this)

Basically, anything that we can show on a flowchart is a control structure.


Have all items in sequence had their turn?
Yes
No
Assign next item to loop variable
Execute all statements in the loop body

Fairfield University DOLAN School of Business

Python for Analytics

Debugging, Libraries, Objects, Collections, and Control Structures
RSI 3, 4