Fairfield University DOLAN School of Business

Python for Analytics

Control Structures
RSI Chapters 5, 6, 7, 8, and 9

Learning Objectives

- **Theory:** You should be able to explain ...
  - How modules and functions facilitate code reuse
  - The parts of a function definition
  - The logic of blocks, conditions, and loops
- **Skills:** You should know how to ...
  - `import` and use modules (and functions)
  - Define a function
  - Write if statements in their many variants
  - Write for loops, while loops and repeat loops
  - Look up and use built-in string methods

Modules and Functions

How to organize code for reuse

Python Modules

A module is a `.py` file that contains a number of reusable definitions and statements

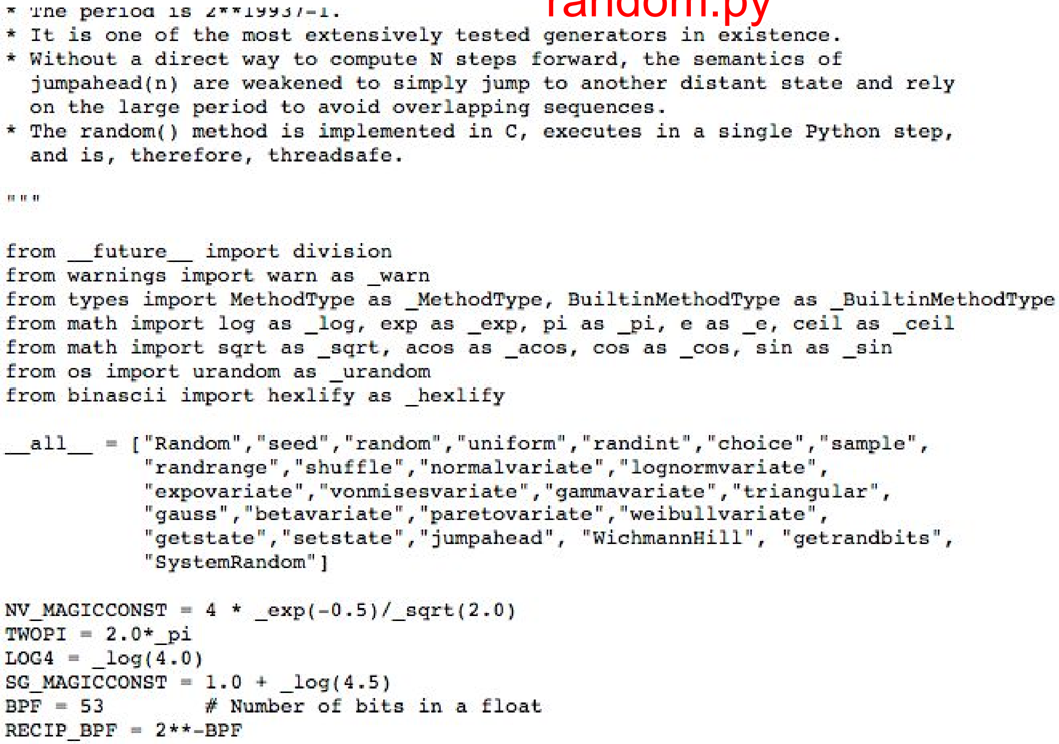
- **Definitions** specify classes (data types) and functions
- **Statements** perform actions like defining variables, calling functions, etc.

We use `import` statements to integrate modules into our code. We then use dot notation to refer to any classes or functions defined by the module.

What’s inside a Module?

Python, of course!

random.py



```
* The period is 2**19937-1.
* It is one of the most extensively tested generators in existence.
* Without a direct way to compute N steps forward, the semantics of
  jumpahead(n) are weakened to simply jump to another distant state and rely
  on the large period to avoid overlapping sequences.
* The random() method is implemented in C, executes in a single Python step,
  and is, therefore, threadsafe.

"""

from __future__ import division
from warnings import warn as _warn
from types import MethodType as _MethodType, BuiltinMethodType as _BuiltinMethodType
from math import log as _log, exp as _exp, pi as _pi, e as _e, ceil as _ceil
from math import sqrt as _sqrt, acos as _acos, cos as _cos, sin as _sin
from os import urandom as _urandom
from binascii import hexlify as _hexlify

__all__ = ["Random","seed","random","uniform","randint","choice","sample",
           "randrange","shuffle","normalvariate","lognormvariate",
           "expovariate","vonmisesvariate","gammavariate","triangular",
           "gauss","betavariate","paretovariate","weibullvariate",
           "getstate","setstate","jumpahead", "WichmannHill", "getrandbits",
           "SystemRandom"]

NV_MAGICCONST = 4 * _exp(-0.5)/_sqrt(2.0)
TWOPI = 2.0*_pi
LOG4 = _log(4.0)
SG_MAGICCONST = 1.0 + _log(4.5)
BPF = 53        # Number of bits in a float
RECIP_BPF = 2**-BPF
```

Comments explaining what it does and how to use it

Import statements to integrate other modules

CONSTANTS that can be reused in our code

… (farther down)
Class and Function definitions

Standard vs Custom vs Third-Party

Libraries (and modules) come in three flavors:

- **Standard Library modules** are built into Python
  - You may need to import them but you can count on them to be installed
- **Custom modules** are written by and for your use
  - Source code kept in a folder (library) on your hard drive
- **Third-Party modules** must be installed in order to use them
  - We can use PIP or Conda to download and install them

Function Definitions

A function definition ***encapsulates*** a block of code into a form that can be ***called*** by other code

Syntax

```
def <name>( <parameters> ):
    <statements>
```

Note the punctuation. The ( ), :, spaces and tabs matter. Your code will break if you get it wrong!

- `def` indicates the start of a function definition
- `<name>` is the name of the function
- `<parameters>` is a list of **input variables**
- The indented block of `<statements>` comprise the body of the function

Try it yourself!

```
def add_two_numbers(x,y):
    z=x+y
    return z
add_two_numbers(1,2)
add_two_numbers(1, “2”)
```

Note that function calls *pass* **arguments** but function definitions *define* **parameters.** Confused yet?

- **Name**: `add_two_numbers`
- **Parameters** `x` and `y` act like variables within the body
- **Local variable** `z` is undefined outside of the function body
- The **`return`** statement completes the function call and (optionally) provides a **value**
- Indentation (tabs in this case) indicates the function body

Unit Testing

A well-written module should have pre-defined tests that we can run to check for bugs

Unit tests are pretty easy to write:

```
def add_two_numbers(x,y):
    z=x+y
    return z
import test
test.testEqual(add_two_numbers(1,2), 3)
test.testEqual(add_two_numbers(1, “2”), 3)
```

Do you see a problem with our function? Run the tests yourself to see.

Blocks, Conditionals, and Loops

Three Universal Control Structures

found in any general purpose control language

Blocks

A **block** of code is just a series of statements that are run in the order given. Indentation is used to indicate blocks *inside* other blocks.

```
x=1            → 1
y=2            → 2
print(x+y)     → 3
   print(x+y)  → IndentationError: unexpected indent
```

Explain the error. Why is this an error?

Conditionals

Conditional execution allows a block to run **only** when given conditions are met.

Typical form is **binary selection**

```
if <boolean expression>:
    <success block>
else:
    <fail block>
```

Each condition is a **boolean expression** that evaluates to `True` or `False`

Try it yourself!

```
x=1
y=2
if x>y:
   print(“x is bigger than y”)
else:
   print(“x is not bigger than y”)
```

Some Useful Variations

Unary Selection

```
if <boolean>:
    <success block>
```

Binary selection adds an `else` clause to **unary selection**.

Chained selection inserts one or more `elif` clauses into **binary selection.** The first clause whose condition is met is executed.

The **default clause** at the bottom executes only if **none** of the clauses above it execute.

Chained Selection

```
if <boolean-1>:
    <block-1>
elif <boolean-2>:
    <block-2>
elif <boolean-2>:
    <block-2>
else:
    <default block>
```

Boolean Expressions

So what counts as a boolean expression?

- Values `0` (`False`) and `1` (`True`)
- Relational comparisons
  - `x == y   # is x equal to y?`
  - `x > y    # is x greater than y?`
  - `x >= y   # is x greater than or equal to y?`
- Logical composites of boolean expressions
  - `((x == 1) or ((x > 10) and not (x == 15))`

Loops Revisited

A loop repeats a block over and over again until a termination condition is met.

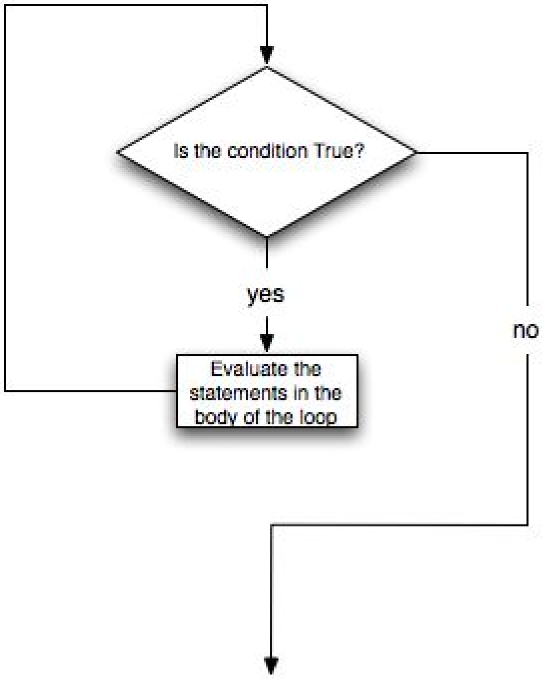
- Our old friend the `for` loop is great for iterating over a set of items
- Sometimes we need to handle more complex logic, which leads us to the while loop

Anything you can do with a `for` loop, you can also do with a `while` loop but not vice versa.

The `while` Loop

```
while <boolean>:
    <statements>
```

- `<boolean>` is a condition to check at the start of each cycle
- `<statements>` in the *body* block are only executed when the condition is `True`



Infinite Loops

Of course, if the `<boolean>` condition always evaluates to True, then the loop repeats forever

```
while 1 == 1:
    print(“U B Ownd!”)
```

Three ways to prevent an infinite loop

- Base the condition on elapsed time or similar *temporal* condition
- Have the body cause the condition to fail eventually
- Use a `break` statement within the body

Strings

A Deep Dive into the
Most Fundamental of all Data Structures

String Literals

String values are said to be ***literals***. When displayed in the interpreter they are always displayed with quotes:

```
x = “ABC”
x                    → ‘ABC’
```

Three kinds of string literals:

- Single quoted literals are like `‘ABC’`
- Double quoted literals are like `“I can say ‘ABC’”`
- Triple quoted literals
```
‘‘‘can span
                    multiple lines.’’’
```

Strings are a Kind of Tuple

We learned about *tuples* in the previous lecture

- Strings are ordered **collections** of characters
- Really, they are just a special kind of ***tuple*** with extra methods and what we call ‘syntactic sugar’ to make them easier to write:
  - “ABCD” is easier than `(‘A’,‘B’,‘C’,‘D’)`
- Two Implications
  - Strings are *immutable*, meaning that you can’t alter a string after it has been created
    - Note: all string methods return *a new string*
  - Strings can be used anywhere tuples can be used

String Operations

All of the things we can do with tuples also apply to strings:

- Concatenation, where we append one string to end of another

  ```
  full_name = first_name + “ ” + last_name
  ```

- Indexing and slicing

  ```
  alpha = “ABCD”
  alpha[1:] → ‘BCD’
  ```

String Methods

RTFM this.
Yes, really.

Strings get all of the tuple methods for free. However, they also come with 16 more built-in methods:

- `upper(), lower(), capitalize()`
- `strip(), lstrip(), rstrip()`
- `center(), ljust(), rjust()`
- `count(),find(), rfind(), index(), rindex()`
- `replace(), format()`

Note: Strings are no more editable than tuples. `upper(), strip()`, etc. just return *new strings.*

String Comparisons

Many of the relational comparisons that apply to numbers also apply to strings.

```
“XYZ” == “ABC”   → False
“ABC” < “XYZ”    → True
“XYZ” > “ABC”    → False
```

String Traversal with for Loops

This works exactly like a tuple, list, or any other collection:

```
s = “ABC”
for c in s:
    print(c)
for i in range(len(s)):
    print(s[i])
```

This should be no surprise since strings are just a special kind of collection.

String Traversal with while Loops

The typical pattern is like this:

```
s = “ABC”
i = 0
while i < len(s):
   print(i,s[i])
   i = i + 1
```

We can, of course, do lots more inside the loop body. This is just an example.

A Digression on `in` (and `not in`)

We have already seen `in` used in for loops

```
for c in s:
   print(c)
```

Technically, the “`c in s`” part is actually a boolean expression, with the `in` operator testing if `c` is *inside* `s`

```
“B” in “ABCD”          →  True
“B” not in “ABCD”      →  False
1 in [2,1,3]           →  True
```

Fairfield University
DOLAN
School of Business

Python for Analytics

Control Structures

RSI Chapters 5, 6, 7, 8, and 9