INSY 6500
Information Systems for Operations

Introduction to Python (but it's still not a programming course)

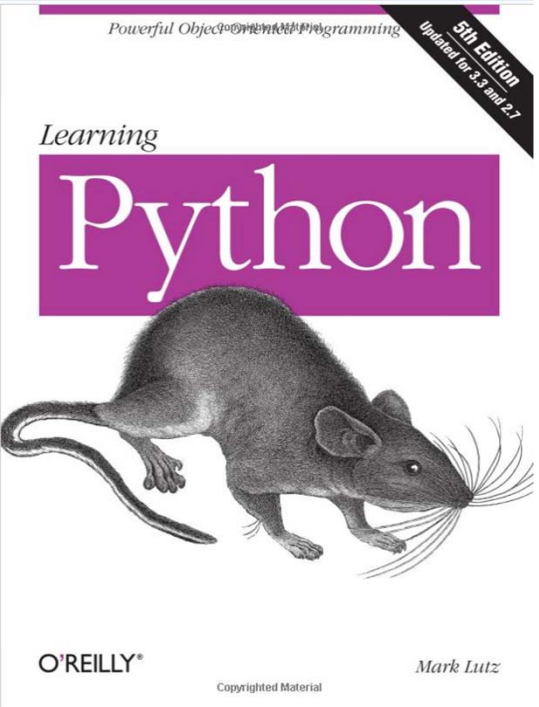
Fall 2019

Jeffrey S. Smith

Industrial and Systems Engineering Department
Auburn University
jsmith@auburn.edu

Other Resources

- Go to google and search on “python 3 tutorial” – 165,000,000 results last time I tried.
- Stack Overflow
- …



So what are the Top Ten Languages of 2018, as ranked for the typical IEEE member and *Spectrum* reader?

Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C++		98.4
3. C		98.2
4. Java		97.5
5. C#		89.8
6. PHP		85.4
7. R		83.3
8. JavaScript		82.8
9. Go		76.7
10. Assembly		74.5

https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

How to "Use" Python

- Python is an *interpreted* language.
- Python tools that we will use:
  - "python.exe"
    - Script Mode
    - Interactive Mode
  - IPythyon
  - Jupyter Notebook*

Example 1:

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

Example 2:

```
for i in range(5):
    j = i + 1
    print ("[{:}] Hello World!".format(j))
```

*Jupyter Notebook is a more general tool that just for Python – as we will see.

Python Object Types

- Python programs
  1. Programs are composed of *modules*
  2. Modules contain *statements*
  3. Statements contain *expressions*
  4. Expressions create and process *objects*

"An *expression* in a programming language is a combination of one or more explicit values, constants, variables, operators, and functions that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce ('to return', in a stateful environment) another value. This process, as for mathematical expressions, is called *evaluation*."
(*Wikipedia - https://en.wikipedia.org/wiki/Expression_(computer_science)*)

Lutz, 2013, pp.93-94

Python Object Types

- Python programs
  1. Programs are composed of *modules*
  2. Modules contain *statements*
  3. Statements contain *expressions*
  4. Expressions create and process *objects*

- Traditional *pillars* (of programming):
  - Assignment
  - Sequence
  - Repetition
  - Selection

```
total = 0
count = 0
while numbers remain
    total = total + next number
if count > 0
      avg = total/count
else
      error
```

Lutz, 2013, pp.93-94

Initial Core Python Statements

3.0.0 Python.ipynb

- Assignment
  - variable = expression
    - `x = 12`
    - `y = x+14`
    - `z = [1, 2, 3]`

- Repetition
  - For Loop

```
for j in [1, 2, 3, 4, 5]:
    statements

for j in range(10)
    statements
```

- Selection
  - If Statement

```
if condition:
    statements
else:
    statements
```

```
if x == 12:
    y = y +1

if x in [1, 2, 3, 4, 5]:
    y = 35
else:
    y = 50
```

Core Data Types

Table 4-1. Built-in objects preview

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'a\x01c', u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes (Part IV, Part V, Part VI)
Implementation-related types	Compiled code, stack tracebacks (Part IV, Part VII)

- Literal

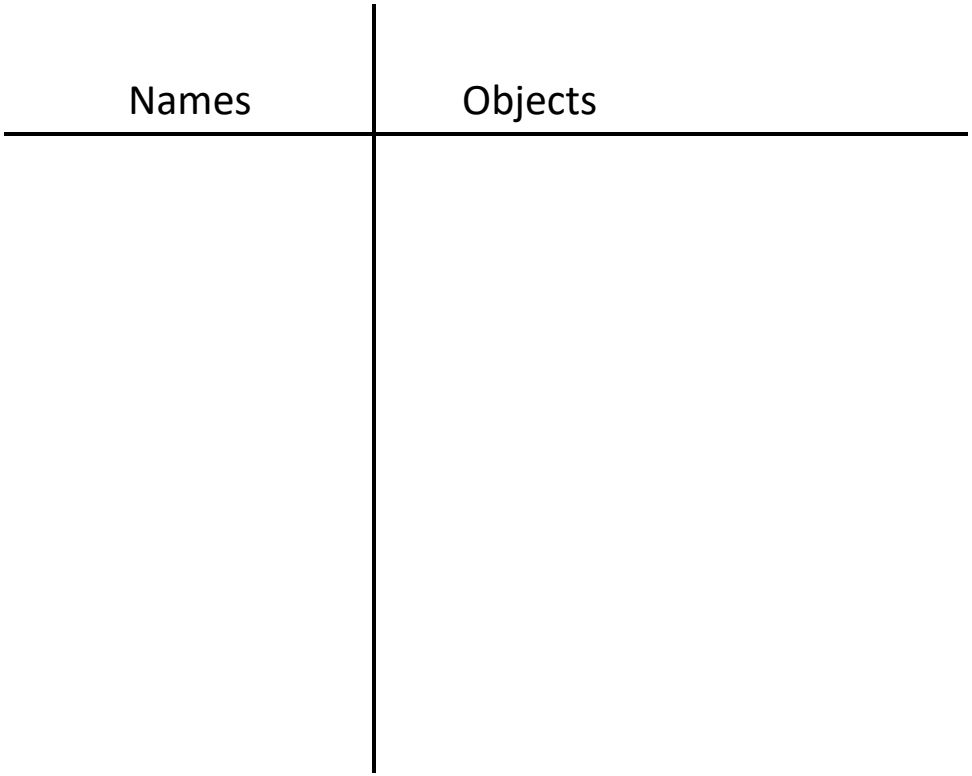
  'In this book, the term literal simply means an expression whose syntax generates an object—sometimes also called a constant. Note that the term "constant" does not imply objects or variables that can never be changed (i.e., this term is unrelated to C++'s const or Python's "immutable"—a topic explored in the section "Immutability" on page 103).'

In Python 3, **`range`** is also a data type.

Lutz, 2013, p.96-8

General Python Concepts

- Dynamically Typed (vs. Statically Typed)
  - Python keeps track of the "type" of a given object and there is no need to declare.



In concrete terms:

1. *Variables* are entries in a system table, with spaces for links to objects;
2. *Objects* are pieces of allocated memory, with enough space to represent the values for which they stand; and
3. *References* are automatically followed *pointers* from variables to objects.

Lutz, 2013, p.183

General Python Concepts

- Strongly Typed (vs. Weakly Typed)
  - The type of a given object does not change and you can only perform operations that are valid for the object's type.
- Mutable vs. Immutable
  - numbers, strings, and tuples are *immutable*
  - lists, dictionaries, and sets are not immutable (i.e., they are *mutable*)
- Namespaces
  - Each module has its own namespace

Numbers

- Integers – No fractional component
- Floating point – Fractional component
- Others (later)
- Build-in Numeric Tools (p. 136)
  - Expression Operators (+, -, *, /, >>, **, &, etc.)
  - Built-in mathematical functions (pow, abs, round, int, ceiling, floor, etc.)
  - Utility modules (random, math, etc.)

Strings

- sequence – “A positionally ordered collection of other objects”
- Sequence operations
  - Indexing
  - Slicing
  - Concatenation
  - Repetition
- Immutability (revisited)
- Type-specific operations
  - find
  - replace
  - split
  - …

Lutz, 2013, p.99

The `format()` method for string objects

- `format()` creates a new string object and allows the user to control the formatting of the object.

```
a = 'Jim'
b = 'Carl'
c = 'Nancy'
"{}, {}, {} are going on a trip".format(a, b, c)


salary = 122000
"{}'s salary is ${:,.2f}".format(a, salary)
```

Lists

3.1.0 Python Lists and Dictionaries.ipynb

- "Most general *sequence* provided by the language"
  - Ordered collection of *arbitrary* objects
  - Mutable
- Sequence operations (index, slice, concatenate, repetition, etc.)
- Type-specific operations
  - append
  - pop
  - remove
  - sort
  - reverse
  - …

Lutz, 2013, p.109

Introduction to List Comprehensions

- Consider the matrix *m* (the list of 3 lists)
  - 2nd row: m[1]
  - 2nd column: ???
- *Comprehension* – "build a new list by running an expression on each item in a sequence, one at a time, from left to right"

```
[r[1] for r in m]   [m[i][1] for i in range(len(m))]
```

Lutz, 2013, p.111

List Examples - People

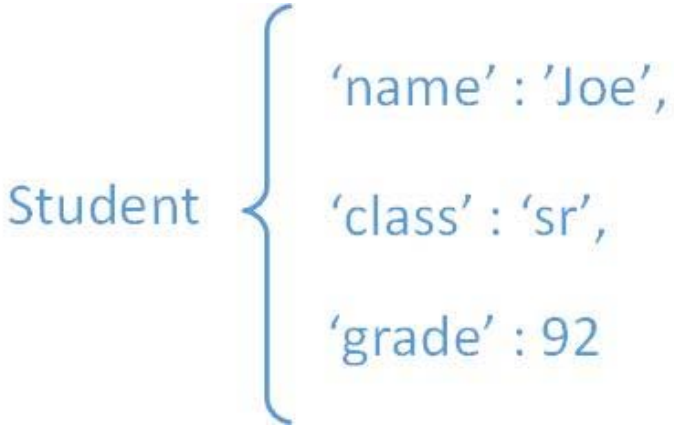
```
# creating a list to define a person
person = ["Tom Howard", 54, 6.0]

# creating a list of lists to define a team
people = [
      ["Tom Howard",          54,  6.0],
      ["Jane Grimm",          19,  4.9],
      ["Sam Brown",           25,  6.2],
      ["Sarah Joan Spade",    26, 5.25],
      ["Blaine Jones",        62,  5.8],
      ["Devin Callahan",      32, 5.92],
]
```

- ***Data Set*** – a set of *data items* (people). Each item (person) has a set of *attributes* (name, age, height)
- ***Data Structure*** – storing the data set in a list of lists.

Dictionaries

- A *dictionary* is a set of *key*, *value* pairs
- Whereas in a list, you use an integer *index*, i.e., l[2] or l[178], in a dictionary, you use the key.




```
student = {'name': 'Joe', 'class':'sr', 'grade':92}
```

Nested Dictionaries

ID	Name	Gender	HW1	HW2	HW3	Exam1	Exam2	FinalExam
b0001	Jane Doe	F	95	87	92	88	93	90
b0002	John Blue	M	55	76	89	77	82	80
b0003	Kim Tester	F	80	75	65	70	75	80
b0004	Larry Black	M	90	90	92	95	85	94
b0005	Susan White	F	65	52	85	45	80	82

```
students = {
  'Jane Doe‘ :   {'ID':'b0001','Gender': 'F','HW1':95,'HW2':87, 'HW3':92,'Exam1': 88,'Exam2':93,'FinalExam':90},
  'John Blue':   {'ID':'b0002','Gender': 'M','HW1':55,'HW2':76, 'HW3':89,'Exam1': 77,'Exam2':82,'FinalExam':80},
  'Kim Tester':  {'ID':'b0003','Gender': 'F','HW1':80,'HW2':75, 'HW3':65,'Exam1': 70,'Exam2':75,'FinalExam':80},
  'Larry Black': {'ID':'b0004','Gender': 'M','HW1':90,'HW2':90, 'HW3':92,'Exam1': 95,'Exam2':85,'FinalExam':94},
  Susan White':  {'ID':'b0005','Gender': 'F','HW1':65,'HW2':52, 'HW3':85,'Exam1': 45,'Exam2':80,'FinalExam':82}
  }
```

Truth Values and Boolean Tests

- All objects have an inherent *Boolean* true or false value.
  - Any nonzero number or nonempty object is *true*.
  - Zero numbers, empty objects, and the special object *None* are considered *false*.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return True or False (custom versions of 1 and 0).
- Boolean *and* and *or* operators return a true or false *operand object*.
- Boolean operators stop evaluating ("short circuit") as soon as a result is known.
- `bool(`*`arg`*`)` is a built-in function to test the Boolean value of an argument (expression)

Lutz, pp. 381

Python Boolean Expression Operators

- `X and Y`
  - Is true if both X and Y are true
- `X or Y`
  - Is true if either X or Y is true
- `not X`
  - Is true if X is false (expression returns True or False)

Lutz, pp. 381

Sets and Tuples

- Sets - A *set* is an "unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory."

```
x = {1, 2, 3, 4}
```

- Tuples - A *tuple* is similar to a list, except that tuples are immutable.

```
t = (1, 2, 3, 4, 'hello', 'there', 5)
```

Lutz, 2013, p.163

Functions

- "In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name. Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts" (Lutz, p. 473).
- Two primary uses:
  - To simplify and improve *your* code
  - To use other code people's code from libraries/modules (like NumPy and Matplotlib)

General Function Structure

```
def name(param1, param2, ..., parmn):
    statements
    statements
    …
    return(x, y, z, ...)
```

Parameters and return values are optional

Files

- File objects are created using the `open()` function.
- We are primarily interested in plain text files.

```
f = open('test.txt', 'w')
```

```
lines = [i.rstrip() for i in open('data1.txt', 'r')]
```

Python Statements

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/finally`	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with/as`	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Deleting references	`del data[k]`
		`del data[1:j]`
		`del obj.attr`
		`del variable`

Lutz, pp. 320-321

Compound Statements

```
Header line:
    Nested statement block
```

```
if Count > 0:
    Avg = float(Total)/Count
    print “Average: {}”.format(Avg)
else:
    Avg = 0
    print “Nothing to average”
```

General `if` Statements

```
if test1:
    statements1
elif test2:
    statements2
elif test3:
    statements3
…
else:
    statementsn
```

- *test* conditions evaluate to True or False.
- `elif` and `else` components are optional.

Special Case: Ternary `if` Statements

```
if test1:
    x = A
else:
    x = B
```

```
x = A if test1 else B
```

General `while` Statements

```
while test:
    statements1
else:
    statements2
```

- *test* conditions evaluate to True or False.
- `statements1` execute repeatedly until test fails (evaluates to False)
- `else` + `statements2` (*optional*) – execute if the loop didn't end with a `break`

General `for` Statements

```
for target in object:
    statements1
else:
    statements2
```

- Iterates through the items in the `object` assigning each value to `target`.
- `else` + `statements2` (optional) – execute if the for didn't end with a `break`

Summary

- "Executing" Python
  - Script mode
  - Interactive mode
- Python program structure
  - Modules
  - Statements
  - Expressions
  - Objects
- Dynamically typed
- Strongly typed
- "Type lives with the object, not the name"
- Namespaces
- Object types:
  - Numbers
  - Strings
  - Lists
  - Dictionaries
  - Files
  - Others …
- Data structures