Python

Lubanovic: Introducing Python

Portland State
Computer Science

Why Python?

- Full-stack course covers NodeJS
- Concise and expressive
- Ability to support multiple programming paradigms (imperative, object-oriented, functional)
  - Both a + and a -
- Rare language used by beginners all the way to experts

- Glue language with bindings for all kinds of code
  - Data science and machine learning
    - Pandas, NumPy: data analysis
    - Jupyter: data visualization
    - PySpark: manipulating data on clusters of computers
    - PyTorch, TensorFlow: machine learning
  - Security
    - IDA Pro, Immunity debugger control, Penetration testing tools
    - angr, Manticore symbolic execution
  - Web development
    - Reddit, Dropbox, Instagram (Django), Google, YouTube
  - API bindings for all Google Cloud services
    - Databases, data processing, machine learning models/engines, platform-as-a-service, etc.

Netflix: Python programming language is behind every film you stream

If you want a job at Netflix, it's probably a good idea to learn programming language Python and all its libraries.

By Liam Tung | April 30, 2019 -- 12:42 GMT (05:42 PDT) | Topic:

According to Python developers at Netflix, the language is used through the "full content lifecycle", from security tools, to its recommendation algorithms, and its proprietary content distribution network (CDN) Open Connect, …

- But, global interpreter lock means you don't want to do HPC within Python

This class

- Assumes you know and have programmed some Python (CS 161)
- Slides go through essential Python
  - Go through them (preferably with the Python interpreter) if you have not programmed in it before
- Will cover a small number of topics that pop up in web application…
  - Basics (Types, variables, numbers, strings, 2 vs. 3)
  - Tuples, Lists, Dictionaries
  - Comprehensions
  - Docstrings
  - Function decorators
  - Classes
  - Modules

Numbers, strings, and variables

Introducing Python: Chapter 2

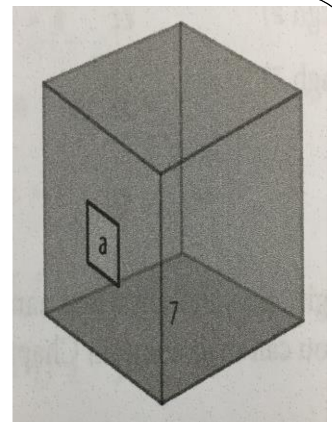
Python types

- Everything is an object
- Conceptually every object is a box with a value and a defined type
- `class` keyword specifies the definition of the object
  - "The mold that makes that box"

- Types built-in to Python
  - boolean (`True` or `False`)
  - integers : `42` or `100000`
  - floats : `3.14159`
  - strings : `"hello"`

- Built-in `type()` method for finding type of an object
  - `type(True)          # 'bool'`
  - `type(42)            # 'int'`
  - `type(3.1415)        # 'float'`
  - `type("foobar")      # 'str'`

Python variables



- Variables name objects
  - Like a post-it note attached to an object
  - Assignment attaches a name, but does *not* copy a value

```
a = 7              # a points to integer object 7
```

  - Assigning one variable to another attaches another post-it note to the object

```
b = a              # b also points to integer object 7
a = 8              # a points to integer object 8
print(b)           # 7
```

Tuples, lists, dictionaries, sets

Introducing Python: Chapter 3

Python tuples

- Immutable, ordered sequence of objects
  - Often used to pass parameters into functions
  - Can have duplicates
  - Denoted with `( )` with `,` separating individual objects in the tuple

    ```
    foo = ( 1 , 3.5 , 'foo' )
    ```
  - Can be indexed by number using the `[ ]` operator

    ```
    type(foo[0])          # int
    type(foo[1])          # float
    type(foo[2])          # str
    ```
  - Immutable

    ```
    foo[2] = 'help'       # TypeError
    ```
  - Single element tuples must include a comma

    ```
    foo = (1)
    type(foo)             # int
    foo = (1,)
    type(foo)             # tuple
    ```

Python lists

- Mutable, ordered sequence of objects
  - Can have duplicates
  - Denoted with [ ] with , separating individual objects in the list

```
empty_list = [ ]
word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
mixed_list = [ 'hello' , 1 , 3.5 , False ]
```

  - Can be indexed by number using the [ ] operator

```
type(mixed_list[0])      # str
type(mixed_list[1])      # int
type(mixed_list[2])      # float
type(mixed_list[3])      # bool
```

Python lists

- Slicing lists (see string slicing)

```
[ <start index> : <end count> : <step size> ]
 word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
 word_list[1:]         # ['quick', 'brown', 'fox']
 word_list[:2]         # ['the', 'quick']
 word_list[::2]        # ['the', 'brown']
 word_list[::-1]
                  # ['fox', 'brown', 'quick', 'the']
```

- Test for the presence of a value with `in` keyword

```
'foo' in word_list      # False
'the' in word_list      # True
```

Python lists

- Sorting list in-place with `sort()` method

```
word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
word_list.sort()
word_list              # ['brown', 'fox', 'quick', 'the']
```

  - Defaults to ascending unless opposite specified via a keyword parameter `reverse`

```
word_list.sort(reverse=True)
word_list              # ['the', 'quick', 'fox', 'brown']
```

- Create a sorted list with built-in `sorted()` function, leave argument alone

```
word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
foo = sorted(word_list)
foo                    # ['brown', 'fox', 'quick', 'the']
word_list              # ['the', 'quick', 'brown', 'fox']
```

Python lists

- Copying lists
  - Recall variables are post-it notes

```
word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
foo_list = word_list
foo_list[0] = 'foo'
word_list        # ['foo', 'quick', 'brown', 'fox']
```

- Copying lists with `copy()` method

```
word_list = [ 'the' , 'quick' , 'brown' , 'fox' ]
foo_list = word_list.copy()
foo_list[0] = 'foo'
word_list        # ['the', 'quick', 'brown', 'fox']
foo_list         # ['foo', 'quick', 'brown', 'fox']
```

Python dictionaries

- Mutable associative array storing **`key:value`** pairs
  - Keys must be unique and immutable
    - Boolean, integer, float, tuple, string
  - Denoted via { } with comma `,` separating individual key-value pairs in dictionary
  - Often used to store JSON
    - Javascript Object Notation (data format typically used as a data transfer format for the web)

Dictionary membership & value extraction

```
bar = {1:2, 'three':4.0}
```

- Test for membership with `in`

```
"three" in bar   # True
"five" in bar    # False
```

- Get a value with `[key]`

```
bar[1]   # 2
bar[5]   # KeyError
```

Dictionary membership & value extraction

```
bar = {1:2, 'three':4.0}
```

- Get all keys using `.keys()`

  ```
  bar.keys()    # dict_keys([1,'three'])
  ```

- Get all values using `.values()`

  ```
  bar.values() # dict_values([2, 4.0])
  ```

- Get all key-value pairs as a list of tuples using `.items()`

  ```
  bar.items()   # [(1,2),('three',4.0)]
  ```

Control Flow, Comprehensions

Chapter 4 (Part 1)

Code syntax

- Whitespace matters
  - Code blocks delineated by indentation level (usually spaces)
  - Generally escape new-line with \

```
>>> 1+2\
... +3
6
>>>
```

  - Parameters can be split without \

```
>> Person(name = "Samuel F. B. Morse",
          Occupation = "painter, inventor",
          Hometown = "Charleston, MA",
          Cool_thing = "Morse code!")
```

Conditionals

- Comparisons with `if`, `elif`, and `else`
  - Statements must end with a colon at the end but no parenthesis needed

```
if n % 4 == 0:
    print("divisible by 4")
elif n % 3 == 0:
    print("divisible by 3")
elif n % 2 == 0:
    print("divisible by 2")
else:
    print("not divisible")
```

Equality in Python

- Two ways `is` and ==
  - `is` checks to see if the objects are the same
    - "Shallow" or "referential" equality
    - Post-it notes attached to same box
    - Intuitively ("a is b")
  - == checks if the bit patterns stored inside the object are the same
    - “Deep” or “structural” equality
    - Intuitively (“a equals b”)
  - For any type, referential equality implies structural equality
  - For mutable types, structural equality does not imply referential equality
    - e.g. copies of list made with the `.copy()` method

Loops

- Iterate with a `for` or a `while` loop

```
for n in [1,2,3,4]:  # Iterate over any iterable
    print(n)
```

- Use `range()` to create a sequence
  - Generator function that returns numbers in a sequence.
  - Acts just like slices, it takes three arguments
    `range(<start>,<stop>,<step>)`

```
range(0,3)          # 0,1,2
range(3,-1,-1)      # 3,2,1,0
```

  - Used with loops

```
for n in range(1,5):
  print(n)
```

  - Equivalent to traditional loop in C-like languages, but easier on branch prediction.

```
i = 1
while i < 5:
  print(i)
  i += 1
```

Comprehensions: Definitions

- An iterable is any object capable of returning its members one at a time
  - Must have the special method `__next__` defined
- An expression is a piece of syntax which can be evaluated to a concrete value.

```
4+lg(8)
7
"Hello" + " world"
x + y  (where values x and y assigned)
```

  - All functions that don't have side effects are expressions
- "Composite" or "compound" data types are those built up from primitive data types and/or other composite data types (e.g. tuples, lists, dictionaries, sets…)

Comprehensions

- A **comprehension**
  - A combination of an expression, an iterable, and potentially a conditional that returns a new compound or composite data type
    - Applies expression to each item in the iterable if condition passes
- Instead of bringing data to the code of a `for` loop…
  We bring the code of a `for` loop to the data

```
for i in range(0,4):
    arr[i] += 1
```

```
arr = [1,2,3,4]
arr = [ x+1 for x in arr ]
```

  - Expression is x+1
  - Iterable is the list [1,2,3,4]
  - The list comprehension of x+1 and list [1,2,3,4] takes the function x+1 and maps it over each item in the list [1,2,3,4] ⇒ [2,3,4,5]
  - Will see this pattern in distributed data processing pipelines
- Syntax support in Python for all mutable, composite data types
- When to use one versus other? It depends…
  - https://leadsift.com/loop-map-list-comprehension/

List Comprehensions

[expression for item in iterable]

```
>>> ns = [n for n in range(1,6)]
>>> ns
[1, 2, 3, 4, 5]
>>> word = "letters"
>>> [ord(ch) for ch in word]
[108, 101, 116, 116, 101, 114, 115]
>>> [chr(ord(ch)-0x20) for ch in word]
['L', 'E', 'T', 'T', 'E', 'R', 'S']
```

[expression for item in iterable if condition]

```
>>> ns = [n for n in range(1,6) if n % 2 == 0]
>>> ns
[2, 4]
```

Functions

Chapter 4 (Part 2)

Positional arguments

- Positional arguments are bound to parameters based on their position in the function call (as with many languages)

```
>>> def power(b,n): return b**n
power(2,3) # 8
power(3,2) # 9
```

Keyword Arguments

- Keyword arguments (`kwargs`) allow us to label each argument passed to the function with the name of the parameter variable.
- Order no longer matters

```
>>> def power(b,n): return b**n
power(b=2,n=3)   # 8
power(n=3,b=2)   # 8
```

Docstrings

- A string at the beginning of a function's body can be pulled out as documentation
- Best practice for any non-trivial function

```
def power(b,n,pp=True):
    """With default param pp set to True,
      will pretty print b**n  Otherwise
      returns the integer value b**n"""
  ret = b**n
  if pp is True:
     print("{} to the power of {} = {}".format(b,n,ret))
  else:
     return ret

>>> help(power)
```

```
Help on function power in module __main__:

power(b, n, pp=True)
    With default param pp set to True, will pretty print b**n
    Otherwise returns the integer value b**n
(END)
```

Docstrings

- Used to automatically generate documentation via `pydoc`
  - This will be graded in your homework submissions
- Example
  - Description of function, its parameters, its return values and exceptions given

```
def connect(url,username,password):
    """This function connects to the specified URL authenticated
       with the provided username and password
    :param url: The URL to connect
    :param username: The username for the authentication
    :param password: The password for the authentication
    :return: Response object of the connected URL
    :raises: HTTP Error when the connection cannot be made.
    """
    try:
        response = requests.get(url, auth=(username,password))
        return response
    except:
        print("This site cannot be reached")
        sys.exit(1)
```

Functions as First-Class Objects

```
>>> def run_something(fun): fun()

>>> def answer(): print(42)

>>> def hi(): print("hello")

>>> run_something(answer)
42

>>> run_something(hi)
hello
```

Nested functions

- In C all functions must be top-level functions
- In Python they do not
  - Allows you to locally scope a function to avoid namespace pollution

```
>>> def outer(a):
        mod = 3
        def inner(b):
            return(a*2 % mod)
        return(inner(a))
...
>>> outer(5)
1
>>> inner(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'inner' is not defined
```

Decorators

- A decorator is a function that takes one function, as an input and returns a wrapped version of it
  - Nested function calls original, is returned to caller for subsequent use

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
...
>>> def add_ints(a,b): return a + b
...
>>> decorated_add_ints = document_it(add_ints)
>>> decorated_add_ints(3,5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
>>> decorated_add_ints(a=5,b=2)
Running function: add_ints
Positional arguments: ()
Keyword arguments: {'a': 5, 'b': 2}
Result: 7
7
```

Decorators

- Decorators have special syntax (`@`)
- Can be used when a function is defined if you only want the decorated version

```
>>> @document_it
... def add_ints(a,b): return a + b
...
>>> add_ints(3,5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

Decorators

- Often used when you have operations that must be run upon every invocation of a function
  - Repeated setup and teardown procedures
  - Timing performance and instrumentation
  - Checking argument types (assertions)
  - Concurrency management (ensuring locks obtained)
  - Ensuring only authenticated access
    - See Python REPL @login_required example at https://www.youtube.com/watch?v=AnNHVupZi5c
  - Python/Flask route definitions

Modules, Packages, & Programs Tools

Chapter 5

Chapter 12

Python library/package support

- Many useful Python packages across domains
- Python package manager called 'pip' (Pip Installs Packages)
  - Equivalent to Node.js `npm`
  - Install packages with `pip install <package name>`
  - Searches the ~39,000 packages in the PyPi repository
  - Example
    - Install Requests package via `pip install requests`
    - Then, can use within program via `import` statement

```
import requests
print(requests.get('http://google.com').text)
```

- Issue
  - Requires administrator access to modify system `python` installation
  - Applications requiring different versions of specific packages

virtualenv

- Creates a local installation of Python for you to create a custom environment so you can install packages specific to your application in it
- Within application directory
  - `virtualenv -p python3 env`
    - Creates a directory called `env` that will create a local installation of python3 that will hold all modules you install (i.e. your environment)
  - `source env/bin/activate`
    - Sets up the shell to use the environment in `env`
    - Must be done everytime you want to run your app
  - `pip install requests`
    - Installs the Requests package into your Python virtual environment
    - Or if multiple packages need to be installed, done via a file `requirements.txt`
      - `pip install -r requirements.txt`
      - Installs all packages specified in file
- Note: to ensure that the `env` directory is not included in your repository we add a `.gitignore` file in the top-level of your repository that includes `env`
  - e.g. `echo "env" >>! .gitignore`

- Then, when wanting to use later…
  - `cd` into directory
  - `source env/bin/activate`
  - … do work …
  - `deactivate (or exit)`

Methods for importing code from packages

- `from` and `import` keywords
  - Inserts library code into namespace of execution environment
  - Multiple options
- Import a specific function (used in flask examples in cs430-src)

```
from xlrd import open_workbook
book = open_workbook("myfile.xls")
```

- Import the package and use the package name as a prefix for each function call

```
import xlrd
book = xlrd.open_workbook("myfile.xls")
```

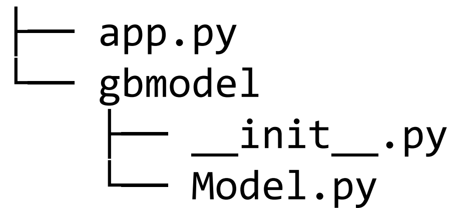
- Import the package with an alias

```
import xlrd as x
book = x.open_workbook("myfile.xls")
```

Make packages from your own modules

- A module is a Python file containing functions or classes
- A package is a collection of modules in a single directory
  - Python identifies a directory as a package if it includes a `__init__.py` file
  - Contents executed when imported into a program (initialization code for package)
- Modules in a package can import other modules in the same package
- Packages can import other packages

Example Guestbook app v2




```python
# __init__.py

def get_model():
    ...
```

```python
# Model.py

class Model():
    ...
```

```python
# app.py
import gbmodel
model = gbmodel.get_model()
```

```python
# app.py
from gbmodel import Model
model = Model()
```

Objects and Classes

Chapter 6

Objects

- Recall
  - Everything in Python is an object.
  - Conceptually every object is a box with a value and a defined type
  - `class` keyword specifies the definition of the object (its type)

- Using the box analogy for Python reference model
  - A class is "the mold that makes that box"
  - An object is a clear, plastic box made from the mold that contains some value

Classes

- Abstract collection of variables and methods

```
class Person():
    def __init__(self, name, email):
        self.name = name
        self.email = email
    def print_contact(self):
        print(self.name, self.email)
```

  - An object is the instantiation of a class
    - `max = Person("Max", "Max@gmail.com")`

- This base class (object type) can be extended to create a new object type

Example

```
class Person():
    def __init__(self, name, email):
        self.name = name
        self.email = email
    def print_contact(self):
        print("Name: {}\n".format(self.name))
        print("Email:{}\n".format(self.email))

class Student(Person):
    def __init__(self, name, email, stu_id):
        self.name = name
        self.email = email
        self.stu_id = stu_id
    def print_id(self):
        print("ID: {}".format(self.stu_id))

max = Student("Max", "max@pdx.edu", "11235")
```

```
>>> max.print_contact()
Name: Max
Email: max@pdx.edu
>>> max.print_id()
ID: 11235
```

- Student extends base class of Person by using Person as a parameter to its class declaration
  - Inherits the methods of Person
- If defined, student's `__init__` method overrides Person's (to add a student ID)
  - Extends class with additional method `print_id`