Python

Lubanovic: Introducing Python
Why Python?

- Full-stack course covers NodeJS
- Concise and expressive
- Ability to support multiple programming paradigms (imperative, object-oriented, functional)
  - + and -
- Rare language used by beginners all the way to experts
- Good for small web applications
Why Python?

- Widely used glue language
- Python 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
    - angr, Manticore symbolic execution
    - Penetration testing tools
  - 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.)
- 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 162)
- Slides go through essential Python
  - Go through all of 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, Sets
  - Comprehensions
  - Positional vs. Keyword Arguments
  - Docstrings
  - Function decorators
  - Classes
  - Underscores
  - 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)`
- `type(42)`
- `type(3.1415)`
- `type("foobar")`
Python variables

- Variables name objects
  - Assignment attaches a name
  - Does *not* copy a value
    ```python
    a = 7  # a points to integer object 7
    ```
- Variable names are like a post-it notes attached to an object
- Assigning one variable to another attaches another post-it note to the object
  ```python
  b = a   # b also points to integer object 7
  a = 8   # a points to integer object 8
  print(b)  # 7
  ```
Python numbers

- Integers
  - In Python 3, integers are "big" (in Python 2, int are 32-bit and long are 64-bit)
  - No more truncation and overflow errors (but at a cost...NumPy native numbers)
    
    ```python
    print(10**100)
    ```
Python strings

- Immutable sequence of characters
- In Python 3, strings decoupled from their encoding (in Python 2, strings are ASCII)
- Once defined, they cannot be changed in-place
Python strings

- Created via 1 or 3 single quotes or double quotes
  
  `'foo' == '''foo''' == "foo" == """foo""

- Triple quotes are multi-line
  
  '''foo
      bar'''

- Allows you to minimize escaping
  
  'In Edgar Allan Poe\'s poem "The Raven"'

  '''In Edgar Allan Poe's poem "The Raven"'''

- Converting to strings from other types
  
  `str(True)`
  `str(99.5)`
  `str(1.0e4)`
  `str(99)`

- Note, the built-in function `print()` will call `str` on any non-string parameter automatically
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 list
    ```python
    foo = (1, 3.5, 'foo')
    ```
  - Can be indexed by number using the [ ] operator
    ```python
type(foo[0]) # int
type(foo[1]) # float
type(foo[2]) # str
    ```

- **Immutable**
  ```python
  foo[2] = 'help' # TypeError
  ```

- Single element tuples must include a comma
  ```python
  foo = (1)
type(foo) # int
  foo = (1,)
type(foo) # tuple
  ```
Python lists

- **Mutable**, ordered sequence of objects
  - Can be indexed by number using the `[ ]` operator
  - Can have duplicate objects
  - Denoted with `[ ]`
  - Individual objects in the list separated by a comma,
Python lists

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

- Indexing lists
  
  ```python
  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)
  \[ \text{<start index>} : \text{<end count>} : \text{<step size>} \]
  
  \[
  \text{word_list} = [ \text{'the'}, \text{'quick'}, \text{'brown'}, \text{'fox'} ]
  \]
  
  \[
  \text{word_list}[1:] \quad \# [\text{'quick'}, \text{'brown'}, \text{'fox'}]
  \]
  
  \[
  \text{word_list}[:2] \quad \# [\text{'the'}, \text{'quick'}]
  \]
  
  \[
  \text{word_list}[:2] \quad \# [\text{'the'}, \text{'brown'}]
  \]
  
  \[
  \text{word_list}[::-1] \quad \# [\text{'fox'}, \text{'brown'}, \text{'quick'}, \text{'the'}]
  \]

- Test for the presence of a value with \texttt{in} keyword

  \[
  \text{'foo'} \text{ in word_list} \quad \# \text{False}
  \]
  
  \[
  \text{'the'} \text{ in word_list} \quad \# \text{True}
  \]
Python lists

- Sorting list in-place with `sort()` method
  ```python
code
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
  ```python
code
word_list.sort(reverse=True)
word_list          # ['the', 'quick', 'fox', 'brown']
```
- Create a sorted list with built-in `sorted()` function, leave argument alone
  ```python
code
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
    ```python
    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
  ```python
  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']
  ```
Dictionaries

- Associative array abstract data type
  - Key-value store
  - Keys must be unique and immutable
    - Boolean, integer, float, tuple, string
  - The dictionary itself is mutable
- Syntax
  - Denoted via \{ \}
  - Comma separated key:value pairs
- Often used to store JSON
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

```python
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)]

• Assign with =, copy with copy()
  foo = bar
  foo    # Points to same object as bar
  who = foo.copy()  # New dict created
```
Sets

- “A set is like a dictionary with its values thrown away, leaving only the keys.” - Lubanovic
- Create empty set with `set()`
- Create a populated set with `{1, 2, 3, 4}`
- Conversion from other data types
  
  ```
  set("letters")  # {'l','e','t','r','s'}
  ```
- Implemented as a dictionary with null values in keys
Control Flow, Comprehensions

Chapter 4 (Part 1)
Code syntax

- Whitespace matters... most of the time
  - Code blocks delineated by indentation level (usually spaces)
  - Generally escape new-line with \
    ```python
    >>> 1+2\n    ... +3
    6
    >>>
    ```
- Parameters can be split without \
  ```python
  >>> Person(name = "Samuel F. B. Morse",
          Occupation = "painter, inventor",
          Hometown = "Charleston, MA",
          Cool_thing = "Morse code!"
  )
  ```
Conditionals

- Comparisons with if, elif, and else
- You need a colon at the end but no parenthesis

```python
if n % 4 == 0:
    print("divisible by 4")
ellif n % 3 == 0:
    print("divisible by 3")
ellif 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
    - Intuitively ("a is b")
    - Post-it notes attached to same box
  - `==` checks if the bit patterns stored inside the object are the same
    - “Deep” or “structural” equality
    - Intuitively (“a equals b”)
- For immutable types, these are equivalent
- For mutable types, they are not
- For any type, referential equality implies structural equality
Loops

- Iterate with a `for` or a `while` loop
  
  ```python
  for n in [1,2,3,4]:  # Iterate over any iterable
    print(n)
  ```

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

- Use `range()` to create a sequence

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

  - Also just like slices, the last value created will be just before the index of stop
    
    ```python
    range(0,3)  # 0,1,2
    range(3,-1,-1)  # 3,2,1,0
    ```

  ```python
  for n in range(1,5):
    print(n)
  ```
“Composite” or “compound” data types are those built up from primitive data types and/or other composite data types.

In Python, an iterable is any object capable of returning its members one at a time:
- Must have the special method `__next__` defined.
- All the built-in composite data types are iterables.

An expression is a piece of syntax which can be evaluated to a concrete value.

- $4 + \log(8)$
- $7$
- "Hello" + " world"
- $x + y$ (where values $x$ and $y$ assigned)
- All functions that don’t have side effects are expressions.
Comprehensions

- A **comprehension**
  - A combination of an *expression*, an *iterable*, and potentially a *conditional*
  - Applies expression to each item in the iterable if condition passes
    - Creates a new object
- Instead of bringing data to the code in a for loop, we bring the code in a for loop to the data

```python
arr = [1,2,3,4]
for i in range(0,4):
    arr[i] += 1
```

- Expression is `x+1`
- Iterable is the list `[1,2,3,4]`
- The 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] \Rightarrow [2,3,4,5]`

- Syntax support in Python for all mutable, composite data types
  - list comprehensions
  - dictionary comprehensions
  - set comprehensions
  - generator comprehensions
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]

>>> [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)

```python
>>> 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**

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

We can specify parameters to have a default value if not included in the function call

```python
>>> def power(b,n,pp=True):
    # pp (pretty print)
    ret = b**n
    if pp is True:
        print(f"{{}}{{}}={ret}\n".format(b,n,ret)
    else:
        return ret

power(2,3)  # 2 to the power of 3 = 8

power(2,3,pp=False)  # 8

power(2,3,pp=True)  # 2 to the power of 3 = 8
```
Positional Arguments via tuple

- A single asterisk in front of the parameter name will be read as a tuple of positional arguments
- By convention named *args

```python
>>> def power(b, n):
...     return b**n
>>> def power_args(*args):
...     return args[0]**args[1]

>>> power_args(2, 3)
8
>>> power_args(2, 3, 4)  # OK as tuple, but 4 not used
8

>>> power(2, 3, 4)  # Not OK as positional arg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power_1() takes 2 positional arguments but 3 were given
```
Keyword Arguments via dict

- Two asterisks in front of a parameter name will be read as a dictionary of named arguments
- By convention named **kwargs
- Quotes are not needed around the keys

```python
>>> def power_kwargs(**kwargs):
    return kwargs["b"]**kwargs["n"]

>>> power_kwargs(b=2,n=3)
8
>>> power_kwargs(b=2,n=3,foo="bar")
8

>>> power_kwargs(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power_kwargs() takes 0 positional arguments but 2 were given
```
Docstrings

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

```python
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
```
Docstrings

- Used to automatically generate documentation via `pydoc`
- Example (for you to emulate in your homework)

```python
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

- In Python, functions are first-class objects

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

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

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

>>> 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

```python
>>> def outer(a):
    mod = 3
    def inner(b):
        def inner(b):
            return(a*2 % mod)
        return(inner(a))
    return(inner(a))

>>> outer(5)
1
```
Decorators

- A decorator is a function that takes one function, as an input and returns a wrapped version of it

```python
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
  ```python
  >>> @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
  ```

- Typically used when you have operations that must be run upon every invocation of multiple functions
  - Repeated setup and teardown procedures
  - Timing performance and instrumentation
  - Checking argument types (assertions)
  - Concurrency management (ensuring locks obtained)
  - Ensuring only authenticated access
    - Guarantee secure function access in Web Security class
    - See Python REPL web app, Python/Flask route definitions
Modules, Packages, & Programs

Chapter 5
Python library/package support

- Many useful Python packages across domains
  - Pandas, NumPy: data analysis
  - Jupyter: data visualization
  - PySpark: manipulating data on clusters of computers
  - Requests: HTTP
  - TensorFlow: machine learning
- A user testimonial of the Requests library

abstraction.

Matt DeBoard—
I'm going to get Kenneth Reitz's Python requests module tattooed on my body, somehow.
The whole thing.

Daniel Greenfeld—
Nuked a 1200 LOC spaghetti code library with 10 lines of code thanks to Kenneth Reitz's re-
Importing code from packages

- **from** and **import** keywords
  - Inserts library code into namespace of execution environment
  - Assuming you have already done `pip install <package>` or module files located in same directory (Flask examples)

- Import a specific function
  ```python
  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
  ```python
  import xlrd
  book = xlrd.open_workbook("myfile.xls")
  ```

- Import the package with an alias
  ```python
  import pandas as pd
  d = {'col1': [1, 2], 'col2': [3, 4]}
  df = pd.DataFrame(data=d)
  ```
Imports and namespace pollution

- Imports can lead to conflicts

- Anti-pattern: Import all the names in a package
  ```python
  from xlrd import *
  book = open_workbook("myfile.xls")
  ```

- Example: math and cmath both have functions named log10()
  - Importing everything from both packages would cause namespace pollution
    ```python
    from math import *
    from cmath import *
    ```
  - Which log10()?
  - What if you needed both?
    - Can rename them on import
      ```python
      from math import log10 as log10
      from cmath import log10 as c_log10
      ```
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
  ```python
  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: {}
".format(self.name))
        print("Email: {}
".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")

- 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 overwrites Person’s, so we have to re-declare member variables.
The role of self

- Example
  ```python
  >>> class Person:
  ...     def __init__(self, name, email):
  ...         self.name = name
  ...         self.email = email
  ...     def print_contact(self):
  ...         print(self.name, self.email)
  ...  
  >>> max = Person("max", "max@mail.com")
  >>>
  >>> max.print_contact()
  max max@mail.com
  ```

- The member functions of Person are not stored in each object, but stored once with the class
- When we call `max.print_contact()`
  - The interpreter looks up the class `Person`
  - And passes the object `max` into the class `Person` as `self`