Today

- Python Crash Course
- HW 1 Overview

Assignments

- HW1 (out, due Tues)
Python Crash Course

Data Types in Python

- Primitive types: integers, floats, strings, Booleans
  - Python 3 supports “big” (essentially unlimited size) integers and strings
  - Strings as either 'abc' or "abc" (so that 'a''b''c', 'a'b'c")
- Collection types: list, dicts, tuples (more later)
- User-defined classes (more later)

Variables: variables are "untyped" in that ...

- you don’t declare the type of a variable
- a variable can hold different types of values at different times
- this means no “static” type checking (issues emerge at runtime)

```
x = 42          # x holds an int
y = x + 1       # y is now 43
x = 'ab'        # x now holds a string
y = x + 'c'     # y now holds a string
```
Explicit type conversion (casting)

u = str(5)  # u holds '5'
v = float(5) # v holds 5.0
w = int('42')  # w holds 42
x = bool(0)   # x holds False
y = int('ab')  # raises exception

Implicit type conversion

x = 3 + 0.14   # x holds float 3.14
y = 4 + '2'    # raises exception
print(5)       # same as print(str(5))
List, Dict, and Tuple examples

lists are “heterogeneous” ... can contain values with different types

```python
xs = []  # empty list
n = len(xs)  # n is 0
xs = [1, 2, 3]  # 3 elem list (all ints)
x = xs[0]  # x is 1
x = xs[3]  # raises exception
ys = ['a', [1,2], True]  # 3 elem list (different typed values)
y = ys[1][0]  # y is 1
ys.append(5)  # adds 5 to end of ys
```

dicts store key-value pairs of the form \{k1:v1, k2:v2, ..., kn:vn\}

```python
d = {'name': 'joe', 'age': 42}  # 2 item dict
n = d['name']  # n is 'joe'
d = {42: ['alice', 'joe'], 40: ['bob']}  # people with ages
42 in d  # raises exception
42 in d  # True
d[50] = []  # adds 50:[] to d
d.keys()  # returns "list" of keys
d.values()  # returns "list" of values
d.items()  # returns "list" of pairs (k,v)
```

tuples are fixed-sized lists (can’t add to them once created)

```python
t = ('a', 42, True)  # 3 elem tuple
x = t[0]  # x is 'a'
(x, y, z) = t  # x is 'a', y is 42, z is True
x, y = 'ab', True  # similar idea within parens
```
Basic Syntax:

• whitespace defines code blocks (instead of { }’s)
• all statements in a block have to “line up”
• colons often signal the “body” of statements (like if, while, for, etc)

while loops

```python
x = 0  # outer block
while x < 10:  # outer block
d = 1  # inner block
x = x + d  # inner block
```

functions

```python
def add(x, y):
    t = x + y
    return t

r = add(5, 4)  # r is 9
r = add('a', 'b')  # r is 'ab'
```

if statements

```python
if x < 42:
y = 0
elif x > 42:
y = 1
else:
y = None  # special "null" value
```
for loops: “for var in collection: body”

```python
# prints 10, 20, 30 on separate line
for i in [10, 20, 30]:
    print(i)

# prints 0, 1, 2 on separate line
for i in range(3):
    print(i)

xs = [10, 20, 30]
# prints 0 10, 1 20, 2 30 on separate line
for i in range(3):
    print(i, xs[i])

# more "pythonic"
for i, x in enumerate(xs):
    print(i, x)
```

**Special values**

```python
def stub():
    pass  # basically, do nothing statement

x = None
if x is None:
    # just for demonstration purposes
    pass
elif x is not None:
    x = x + 1
```
Quiz question 4 from last lecture ...

Write a python function (filter) that takes a list (orig_list) and a value (cutoff) and returns a new list (new_list) with all of the values in the original list up to the first occurrence of the value. The new list should be a prefix (i.e., same order of values) of the original list. The values of the original list may or may not be sorted. For example, filter([1,2,3,4], 3) should return [1,2] and filter([4,2,3,1], 1) should return [4,2,3]. If the list is empty or the value isn't in the list, the original list should be returned.

- One answer:

  ```python
def filter(orig_list, cutoff):
    new_list = []
    for x in orig_list:
      if x == cutoff:
        return new_list
      else:
        new_list.append(x)
    return new_list
  ```

- Another answer using a “slice” ([start:stop]) and index function

  ```python
def filter(orig_list, cutoff):
    if cutoff in orig_list:
      return orig_list[0:orig_list.index(cutoff)]
    else:
      return orig_list
  ```
Quiz question 5 from last lecture

In your choice of programming language, write another version of the previous function that solves the problem using recursion (e.g., instead of using a loop or built-in functions).

```python
def filter(orig_list, cutoff):
    if not orig_list or orig_list[0] == cutoff:
        return []
    else:
        return [orig_list[0]] + filter(orig_list[1:], cutoff)
```

Note: “not orig_list” checks if list is empty (bool([]) is False)

Note on comments

- Python has single (`) and multi-line ("""..."""" or '''...''''') comments
- Function comments go below the header

```python
def my_func(p1, p2, p3):
    """Function comment here""
    return p1 + p2 + p3
```
Exceptions: Handle exceptions in `try–except` blocks:

```python
try:
    x = int('fortytwo')
except Exception as e:
    print(e)
```

- `Exception` is the base class for user-defined exceptions
- in this example, the code generates a `ValueError`

```python
def f(x, y):
    try:
        new_x = int(x)
        return new_x + y
    except ValueError:
        print(x, 'not valid')
    except TypeError:
        print(y, 'not valid')
```

```python
f(1,2)  # evaluates to 3
f('a',2)  # prints: a not valid
f(1,'b')  # prints: b not valid
```

Throw an exception using a `raise` statement:

```python
def f():
    raise Exception("uh oh")
```

```python
try:
    f()
except Exception as e:
    print(e)  # prints "uh oh"
```
Python user-defined classes: Basic idea

```python
class ClassName(object):
    # declare class ClassName
    def __init__(self, val=42):
        # constructor, self like "this"
        self.v1 = val
        # declare/init instance var

    def method1(self):
        """method comment""
        # comment
        self.__method2(1)
        # call helper method

    def __method2(self, val):
        # private method
        self.v1 = val
        # set instance var

    def __str__(self):
        # str() call
        s = 'val: ' + str(self.v1)
        return s
```

Creating instances

```python
o1 = ClassName()  # creates object (default arg val)
pprint(o1)  # calls __str__, prints "val: 42"

o2 = ClassName(7)  # creates object (one-arg init)
pprint(o2)  # prints "val: 7"

o3 = ClassName()
o3.method1()
pprint(o3)  # prints "val: 1"
```
Additional Constructs in HW1

Importing and using modules (libraries)

```python
import sys  # loads the sys module
import math # loads the math module (for sqrt function)
```

- To call a library function: `math.sqrt(...)`
- In HW1, `sys` module used for command line args and exiting program

Files ...

```python
f = open(filename, 'r')
```

- Returns a handle to filename for reading (file object)

Accessing file data

- In this class, we only read one character at a time ...
  ```python
  symbol = f.read(1) # read next character, advance position
  ```
- the `tell` function returns the current file position (e.g., `pos = f.tell()`)  
- the `seek` function moves file object to that position (e.g., `f.seek(pos)`)  

Checking for whitespace

```python
symbol.isspace()  # True if space, tab, newline
```

- also: `isalpha()` and `isdigit()`
Raising values to a power

\[ x^{**2} \quad \# \text{ computes } x \text{ squared} \]

Basic string formatting

```python
product = 'jalapeno'
price = 1.05776

print('The product is a %s' % product)
# displays: "The product is a jalapeno"

print('The %s costs $%.2f' % (product, price))
# displays: "The jalapeno costs $1.06"
```

**Exercise:** With a partner, go over `hw1.py` file and try to explain to each other what is going on. Write down any questions you have.