GENERAL ASSEMBLY

INTRO PYTHON PART TWO

REVIEW

- How to Code
- Modules
- Dynamic Typing
- Lambda Functions
- List Comprehensions
- Misc Topics

DYNAMIC TYPING

- Just like with files, our RAM (aka “memory”) is also just sequences of numbers that we must be able to interpret.

DYNAMIC TYPING

- Just like with files, our RAM (aka "memory") is also just sequences of numbers that we must be able to interpret.

- To allow **dynamic typing** (where each name can refer to any object), Python stores a lot of extra information.
- For example, each object stores:
  - **Object's type** (e.g. int)
  - **Reference count** (how many names refer to it)
  - **Value** (often another reference to another part of memory)

DYNAMIC TYPING

C (static typing):

int **a**;

- When we refer to **a**, interpret the memory it refers to as an integer.

Python (dynamic typing):

print(**a**)

- When we refer to **a**, first get the object's type from the memory it refers to. Then, interpret the rest of the memory via the type's rules.

HOW TO CODE

When coding, you only have a limited number of options:

- Use a built-in function.
- Use a method or operator of the object.
- Import a package.
- Write your own function.

MODULES

```
import math
>>> math.sqrt(5)

from math import sqrt
>>> sqrt(5)

from math import *
>>> sqrt(5)
>>> tan(5)
```

LITERALS

Below, we are creating objects without assigning names. These are called **literals**:

- 3 # int object
- [1, 2, 3] # list object
- {1, 2, 3} # set object
- (1, 2, 3) # tuple object

LITERALS

Below, we are creating objects without assigning names. These are called **literals**:

- 3 # int object
- [1, 2, 3] # list object
- {1, 2, 3} # set object
- (1, 2, 3) # tuple object

Can we do the same with functions?

Note that **def f(x)** auto-creates a name **f** in the namespace.

LAMBDA FUNCTIONS

def inc(x): return x + 1

- Creates a function object.
- Assigns the name ‘inc’ to the object.

lambda x: x + 1

- Only creates a function object.
- Intended for a single line.
- Implicitly returns the last evaluated expression.

LAMBDA FUNCTIONS

```
def inc(x): return x + 1
```

- Creates a function object.
- Assigns the name ‘inc’ to the object.

```
lambda x: x + 1
```

- Only creates a function object.
- Intended for a single line.
- Implicitly returns the last evaluated expression.

```
pow(x, y): return x**y
```

```
lambda x, y: x**y
```

LAMBDA FUNCTIONS

inc(2)

(inc)(2) **# (x) means evaluate x first, e.g. 5 * (3 + 4)**

- Evaluating the name inc results in a function object.

(lambda x: x + 1)(2)

- Evaluating the lambda results in a function object.

Interestingly, inc(2) is syntactic sugar for inc.__call__(2)

LAMBDA FUNCTIONS: USES

```
# Square each element of a list.
numbers = [0, 1, 2, 3, 4, 5]
list(map(lambda n: n*n, numbers))

# Sort by ages
people = [['Mike', 40], ['Terry', 20], ['Sarah', 30]]
sorted(people, key=lambda x: x[1])
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100)]
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    if num % 2 == 0:
        cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100) if num % 2 == 0]
```

NOTE you can use these for filtering!

COLUMNAR DATA

```
names = [('Tim', 20),
         ('Sally', 22),
         ('Ryan', 25)]
```

How do we get the first **column** (i.e. a list of names)?

COLUMNAR DATA

```
names = [('Tim', 20),
         ('Sally', 22),
         ('Ryan', 25)]
```

How do we get the first **column** (i.e. a list of names)?

```
names = [person[0] for person in people]
```

ZIP

- Let's go in the opposite direction. We have separate lists of names and ages.
- How do we combine them element-by-element to make a single list of tuples? The easy way is to use zip, which combines each of the first elements, each of the second elements, etc.:

```
names = [‘Tim’, ‘Sally’, ‘Ryan’]

ages = [20, 22, 25]

list(zip(names, ages))

>> [(‘Tim’, 20), (‘Sally’, 22), (‘Ryan’, 25)]
```

```
# Set comprehensions
threes = {n for n in range(0, 1000, 3)}
fives = {n for n in range(0, 1000, 5)}

# Sum of all multiples of three or five
print(sum(threes.union(fives)))
```

FUNCTIONS

- A function allows us to take complex code and refer to it in an easy way, reducing the complexity in our minds.
- For example, "x % 2 == 1" can be tough to understand for beginners. Hence, to make the program easier to read, we refer to what the code does in English, as part of a function call that returns a value:

```
>>> def is_odd(num):
...     return (num % 2 == 1)
```

```
num = 13195

factors = []
for n in range(1,num+1):
        if num % n == 0:
                factors.append(n)

prime_factors = []
for factor in factors:
        n = 1
        while n < factor:
                n += 1
                if factor % n == 0:
                        break
        if n == factor:
                prime_factors.append(factor)

print(max(prime_factors)) # should be 29
```

Hard to Read!

```
def is_multiple(n, m):
        """ Is n a multiple of m? """
        return n % m == 0

def get_factors(num):
        return [n for n in range(1,num+1) if is_multiple(num, n)]

def is_prime(num):
        return True if len(get_factors(num)) == 2 else False

def prime_factors(num):
        return [f for f in get_factors(num) if is_prime(f)]

print(max(prime_factors(13195)))     # should be 29
```

```
# TESTS
print("prime? 2, 3, 4, 41, 111 => ",
      is_prime(2), is_prime(3), is_prime(4),
      is_prime(41), is_prime(111))
print("factors of 36 (contains 6?) => ", get_factors(36))
print("factors of 13195 => ", get_factors(13195))
print("prime factors of 13195 => ", prime_factors(13195))
```

LISTS VS TUPLES

- A list contains ordered data, typically of the same data type:

```
>>> x = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> print(x[0])
```

- A tuple contains ordered groups of variables, often of different data types:

```
>>> x = (“Tim”, 5)   # (name, age) describe one person
>>> print(x)
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

CODING CHALLENGE!

MORE PYTHON YOU MAY ENCOUNTER IN THIS CLASS

INTERACTING WITH THE COMMAND LINE

- Retrieving command-line arguments is easy!

```
import sys
sys.argv    # ARGument Vector (list)

Example: python test.py 1 2 3
sys.argv: ['test.py', '1', '2', '3']
```

INTERACTING WITH THE COMMAND LINE

- Reading from stdin is just like reading a file!

```
import sys
for line in sys.stdin:
    print(line.strip())
```

FILES

```
# ‘with’ ensures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’) as fin:
    for line in fin:
        print(line)

# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

QUOTES

- ‘vs “ ← same – both support escape characters (“\n“)
- “”” ← triple quotes – allows actual newlines

- \ ← if at the end of a non-quoted line of code, allows you to split the line of code (there is an invisible newline after it)

```
>>> names = “Mike Wallace\nClara Simmons”
>>> names.split(“\n”)
>>> names.replace(“\n”, “, “)
```

STRING FORMATTING

```
“Person #{} is {}”.format(2, ‘George’)

“Person #{num} is {name}”.format(num=5, name=‘Henry’)

“Your price will be ${price:.2f}.”.format(price=4.5127)
```

ENUMERATE – WHEN YOU NEED A LOOP INDEX

```
index = 0
for person in people:
        print(“Person #{} is {}”.format(index, person))
        index += 1
```

BECOMES

```
for index,person in enumerate(people):
        print(“Person #{} is {}”.format(index, person))
```

DATES AND TIMES

- datetime.date -> year, month, day
- datetime.time -> hour, minute, second, microsecond,
  tzinfo (TimeZone INFO)
- datetime.datetime -> year, month, day, hour, minute,
  second, microsecond, tzinfo

- datetime.timedelta – difference between dates/times

EXCEPTIONS

```
try:
        num = int('not an int')
except:   # catches ALL exceptions
        print('Exception caught!')


try:
        num = int('not an int')
except ValueError:   # catches the ValueError exception
        print('Exception caught!')
```

EXERCISES!