

Python Course

Ynon Perek

ynon@ynonperek.com



Agenda

- Intro + Short Python Review
- Text Processing
- OS Integration (files / processes)
- Advanced Functions
- Advanced Collections
- Advanced Classes
- Modules: Writing Large

Resources

- Labs:
[https://gist.github.com/ynonpl/
06914f626cd4127899af53a96733157f](https://gist.github.com/ynonpl/06914f626cd4127899af53a96733157f)

History & Culture



- Guido van Rossum - Creator of Python
- Led the project until July 2018
- Today: steering council
<https://www.python.org/dev/peps/pep-8016/>
- Join the party:
<https://discuss.python.org/>

History & Culture

- Oct 2000 - Python2
- 2001 - Python Software Foundation
- Dec 2008 - Python3
- 2010 - Python 2.7.0
- Dec 2016 - Python 2.7.13
- Mar 2017 - Python 3.6.1

Python Vs. The World

Shell Scripts

python2

Java

python3

C#

perl

C++

ruby

Python2 or Python3?

- Strings vs. Bytes
- Lazy Iteration vs. Lists
- Syntax changes (print, input)

Python IDE?

- vim
- pycharm
- Demo

Debugging Python

- Using pycharm
- Using pdb from command line
- Using pdf from within the script
- Using python -i

Command Line pdb

```
$ python -m pdb fib.py
```

Useful pdb commands

- **break (b)** - set a breakpoint
- **continue (c)** - continue with program until breakpoint
- **exit (q)** - abort
- **help (h)** - show all commands
- **list (l)** - show source code around current line
- **return (r)** - continue until current function returns
- **where (w)** - print stack trace
- **restart (run)** - restart the debugged program
- **p** - print value
- **pp** - pretty print value

pdb using set_trace

```
import pdb

def fib(n):
    if n < 2:
        return 1
    pdb.set_trace()
    return fib(n-1) + fib(n-2)

print(fib(10))
```

Debugging via log

```
import logging  
logging.basicConfig(level=logging.DEBUG)
```

Saving log to file

```
logging.basicConfig(level=logging.DEBUG,  
filename='debug.log')
```

Writing Log Messages

```
from __future__ import print_function
import logging

logging.basicConfig(level=logging.DEBUG)

log = logging.getLogger('demo')
log.info('hello world')

logging.info('hello world')
```

More Info:

<https://docs.python.org/2/howto/logging-cookbook.html>



Q & A

Demo: Python Basic Syntax

Lab: Syntax



Text Processing



Agenda

- String manipulation
- Regexp
- CSV

Filter Pattern

```
from __future__ import print_function
import fileinput
import sys

for line in fileinput.input():
    sys.stdout.write(line)
```

fileinput

- reads a list of files line-by-line and allows processing them
- when empty list is passed - use stdin
- when no arg is passed - use sys.argv

inplace editing

```
from __future__ import print_function
import fileinput
import sys

for line in fileinput.input('demo.txt', inplace=True):
    if not line.startswith('#'):
        sys.stdout.write(line)
```

String Related Functions

- str.capitalize
- str.startswith
- str.endswith
- str.index
- str.partition
- str.replace
- str.strip

<https://docs.python.org/2/library/stdtypes.html>

Demo: Simple Calc

```
from __future__ import print_function
import fileinput

total = 0

for line in fileinput.input('demo.txt'):
    cmd, _, val = line.partition(' ')
    if cmd == 'add':
        total += int(val)
    elif cmd == 'sub':
        total -= int(val)
    else:
        raise Exception('Invalid Command: ' + cmd)

print("Result = {}".format(total))
```

Regular Expressions

Finding a pattern
in the text

The Regexp Story

- Started in Mathematics
- 1968 Entered the Unix world through Ken Thompson's qed
- 1984 Standardized by Henry Spencer's implementation

Regular Expressions Today

- Used by all programming languages, including:
 - Php, Tcl
 - Python, Perl, Ruby
 - JavaScript, ActionScript,
 - C#, Java
 - C, Objective C, C++
 - And More



Rule #1

A Simple character matches itself

Examples

Regexp	Meaning
foo	match things with foo: food, goofoo, foodie

Rule #2

A character class matches a single character from the class

Character Class Syntax

- A class is denoted by [...]
- Can use any character sequence inside the squares
[012], [abc], [aAbBcZ]
- Can use ranges inside the squares
[0-9], [a-z], [a-zA-Z], [0-9ab]
- Can use not
[^abc], [^0-9], .

Examples

Regexp	Meaning
[0-9][0-9]	Display things with 2 digits: x11y, abc10d, 102030

Which Ones Match ?

hello [ux][012]

hello world

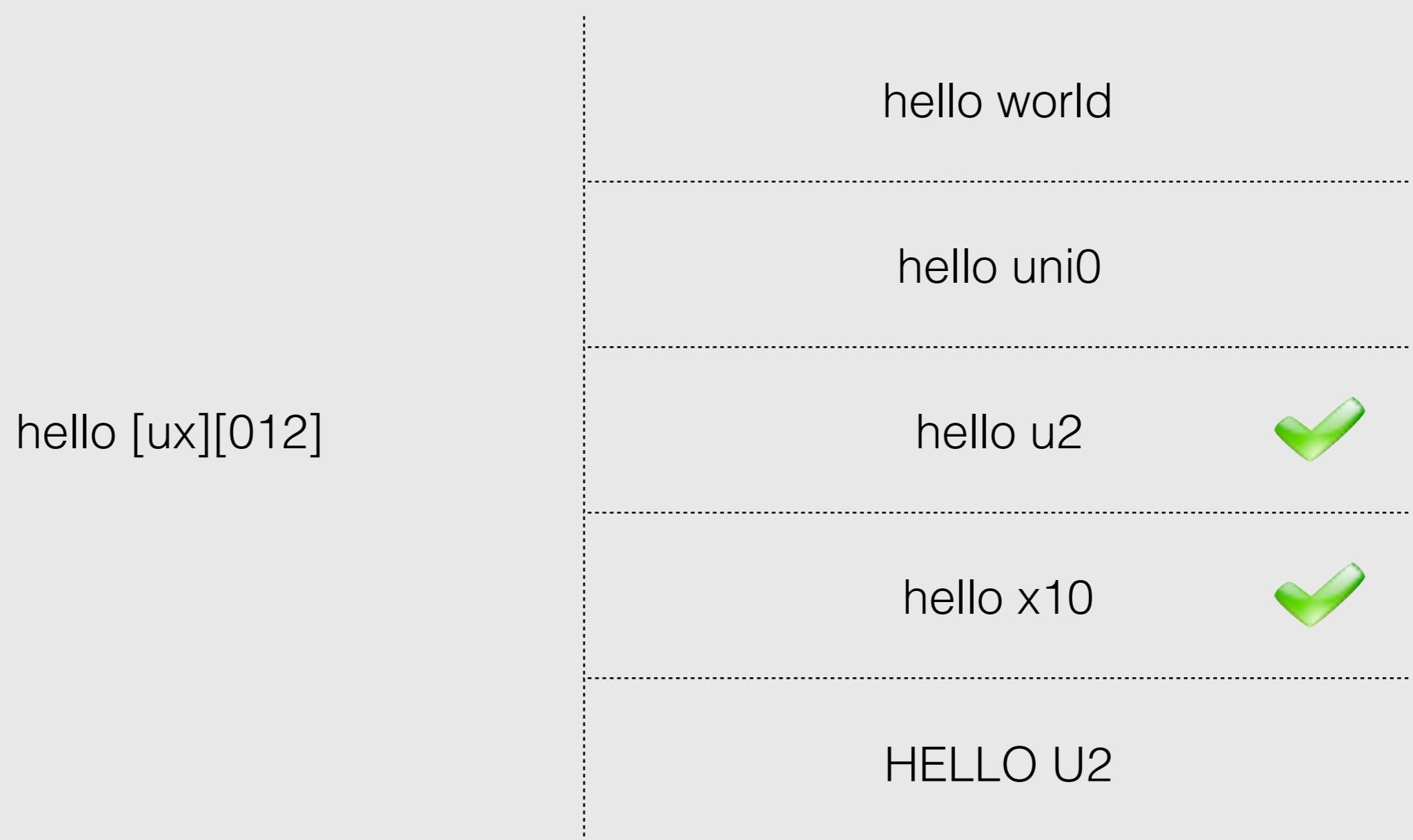
hello unix

hello u2

hello x10

HELLO U2

Answer



Rule #3

A quantifier denotes how many times a letter will match

Quantifiers

- * means match zero or more times - {0,}
- + means match one or more times - {1,}
- ? means match zero or one time - {0,1}
- {n,m} means match at least n but no more than m times
- {n} means match exactly n times

Which of these match ?

[0-9]{2}-?[0-9]{7}

08-9112232

421121212

054-2201121

Phone: 03-9112121

Bond 007

Which of these match ?

[0-9]{2}-?[0-9]{7}

08-9112232



421121212



054-2201121



Phone: 03-9112121



Bond 007

Which of these match ?

(http://)?w{3}\.[a-z]+\.\com

www.google.com

www.ynet.co.il

http://mail.google.com

http://www.home.com

http://www.tel-aviv.com

Which of these match ?

(http://)?w{3}\.[a-z]+\.\com

www.google.com



www.ynet.co.il

http://mail.google.com

http://www.home.com



http://www.tel-aviv.com

Rule #4

An assertion will match on a condition, not capturing input characters

Assertions

- `^` matches the beginning of a line
- `$` matches the end of a line
- `\b` matches word boundary

Which of these match ?

`^d`

`drwxr-xr-x dive`

`-rwxr-xr-x dive`

`lrwxr-xr-x dive`

`drwxr-xr-x /home`

`-rwxr-xr-x /etc/passwd`

Which of these match ?

^d

drwxr-xr-x dive



-rwxr-xr-x dive

lrwxr-xr-x dive

drwxr-xr-x /home



-rwxr-xr-x /etc/passwd

Rule #5

A pipe is used to combine multiple patterns in an or

Or

- Use | for or
- Use (...) for precedence
- Example:

^ (foo | be) d

Regexp + Python

Module re

- `re.compile`
- `re.match`
- `re.search`
- `re.split`
- `re.sub`

Demo: remove comments

```
from __future__ import print_function
import fileinput
import re
import sys

COMMENT = re.compile(r'^#')

for line in fileinput.input():
    if not COMMENT.search(line):
        sys.stdout.write(line)
```

Guidelines

- Use `r'...'` for RE strings
- Use `re.compile` to create static patterns
- Use strings for dynamic patterns

Captures

- `re.search` returns a match object
- That object can contain information about what was matched
- Use `(?P<name>)` to name your captures

Demo: counting shells

```
COMMENT = re.compile(r'^#')
BLANK   = re.compile(r'^$')
SHELL   = re.compile(r'(?P<name>\w+)\$')
shells = {}

for line in fileinput.input():
    if COMMENT.search(line): continue
    if BLANK.search(line): continue
    match = SHELL.search(line)
    if match is None: continue

    name = match.group('name')
    shells[name] = shells.get(name, 0) + 1

print(shells)
```

CSV files

Reading Data from CSV

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Writing to CSV file

```
import csv
with open('some.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```



Q & A

Lab: Text Processing





OS Integration



Why

- Python is a useful “glue” language:
 - reading/writing files
 - running other programs

Module subprocess

- Responsible for calling external processes:
 - Run and wait for result
 - Run and wait for output
 - Run multiple and wait for some to finish

Run and wait for result

```
from __future__ import print_function
import subprocess

proc = subprocess.Popen('ls')
ok = proc.wait()
print('ls returned: {}'.format(ok))
```

Popen with arguments

```
from __future__ import print_function
import sys

import subprocess

proc = subprocess.Popen(['ls', '-l', '-rt'] + sys.argv[1:])
ok = proc.wait()
print('ls returned: {}'.format(ok))
```

Hiding process output

```
from __future__ import print_function
import subprocess

proc = subprocess.Popen(['ls', '-l', '-rt'],
                      stdout=subprocess.PIPE)
ok = proc.wait()
print('ls returned {}'.format(ok))
```

Using grep to find-in-file

```
from __future__ import print_function
import subprocess

proc = subprocess.Popen(['grep', 'bash', '/etc/shells'],
                      stdout=subprocess.PIPE,
                      stderr=subprocess.PIPE)
ok = proc.wait()
if ok == 0:
    print("Found bash in /etc/shells")
else:
    print("bash not found in /etc/shells")
```

Getting Command Output

- The function `subprocess.check_output` runs a command and returns its output
- No need to pipe `stdout`

Getting Command Output

```
from __future__ import print_function
import subprocess

ts = subprocess.check_output(
    [ 'date', '+%s' ],
    stderr=subprocess.STDOUT).strip()

print('Timestamp: {}'.format(ts))
```

Running Multiple Processes

- Call Popen(...) several times before waiting to get all processes working together

Running Multiple Processes

```
import subprocess

files = [
    'http://gallery.kingsnake.com/data/77374crockys\_and\_kittens\_2\_131.jpg' ,
    'http://www.lashworldtour.com/wp-content/uploads/2013/07/two-kittens-fascinated.jpg' ,
    'http://i.dailymail.co.uk/i/pix/2011/12/03/article-2069512-0F09244F00000578-972\_634x526.jpg']
]

procs = [subprocess.Popen(['wget', x], stdout=subprocess.PIPE,
stderr=subprocess.PIPE) for x in files]
results = [x.wait() for x in procs]
```

pexpect

- Use pexpect module to talk to interactive programs

pexpect

```
import pexpect

child = pexpect.spawn('tr a-z b-za')
child.setecho(False)

word = 'hello'
while True:
    child.sendline(word)
    child.expect ('\w+\r\n')
    print child.after
    word = child.after.strip()
    if word == 'hello': break
```

OS Information

- `glob.glob('*.txt')` - gets a glob pattern
- `os.path.join('/home/ynon', 'python', 'examples')`
- `os.walk`

OS Walk

```
import os
from os.path import join, getsize
n = 1

for root, dirs, files in os.walk(os.path.expanduser('~/tmp')):
    if root.startswith('.'): continue
    print("{pad:{n}s} {fname}".format(fname=root, n=n, pad=' '))
    n += 2
    for fname in (f for f in files if not f.startswith('.')):
        print("{pad:{n}s} {fname}".format(fname=fname, n=n, pad=' '))
```

Argument Parsing

- Python has an argparse module which helps play friendly with unix
- We define the arguments and python parses them from command line

argparse demo

- Detect two file extensions and an optional -n

```
from __future__ import print_function
import os
import glob
import argparse

parser = argparse.ArgumentParser(description='A simple file renamer')
parser.add_argument('-n', action='store_true', dest='noop')
parser.add_argument('efrom', help='file extension to convert from ')
parser.add_argument('eto', help='file extension to convert to')
args = parser.parse_args()

# args.noop is True if -n was specified
# args.efrom is the first name
# args.eto is the second name
```

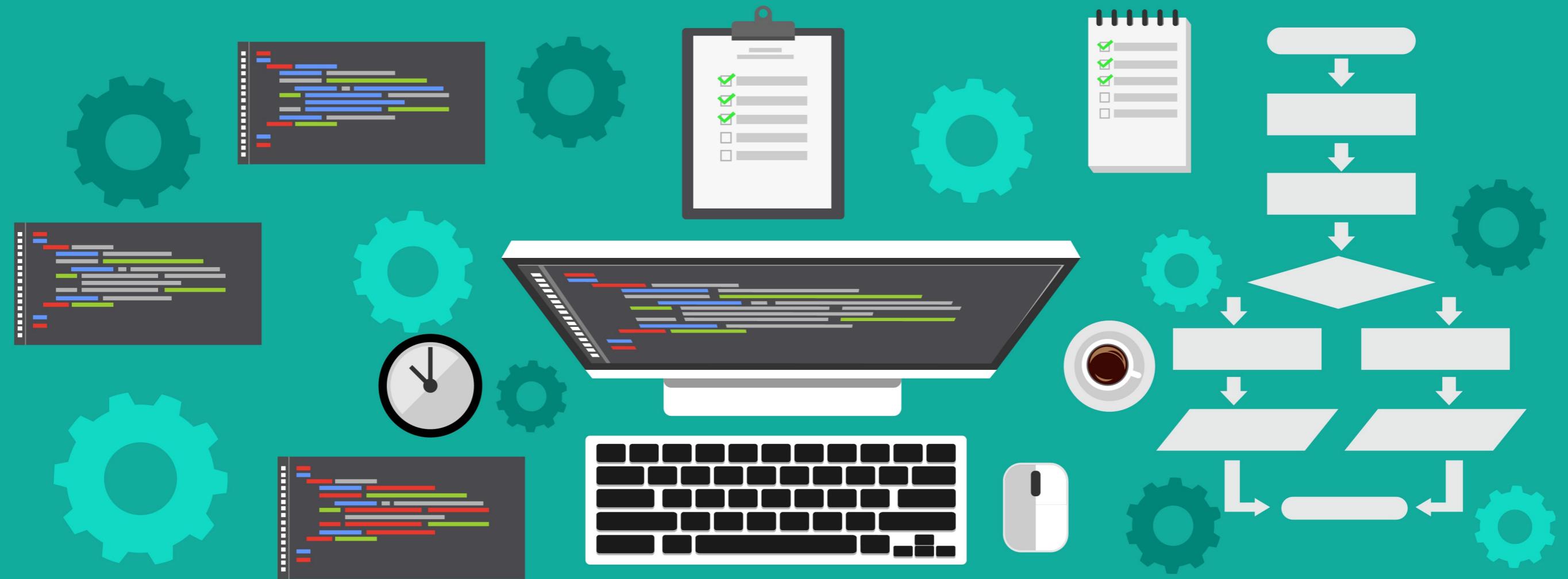


Q & A

Lab: OS Integration



Functions



Hello Functions

```
def function_name(args):
    """ function documentation """
    # function code
    # optional return statement
    return 10
```

Hello Functions

- First class citizens
- Read docstring with help(...)
- Gets positional + keyword arguments

Type Validation

- No built in type validation in python
- Options:
 - type hints
 - assert
 - live with it

Type Hints

```
def lastword(sentence: str) -> str:  
    """ returns the last word from a sentence """  
    return sentence.split().pop()  
  
lastword(10)
```

Assert

```
def lastword(sentence):
    """ returns the last word from a sentence """
    assert type(sentence) == str, 'String argument expected'

    return sentence.split().pop()

lastword(10)
```

When To Use Each

- Try to live with it if possible
- Assert when code is meaningless

Variable Scope

```
from __future__ import print_function
_total = 0

def add(val):
    global _total
    _total += int(val)

add(10)
add('20')
add(5)
print('total: {}'.format(_total))
```

Function Objects

- Functions can be saved in variables or passed as input to other functions

```
# Function Definition
def twice(x):
    return 2 * x

# Function Reference
g = twice
f = twice

# Calling Functions
print(f(g(10)))
```

Lambda Expressions

- Lambda defines an anonymous one-line function

```
# Function Reference
g = lambda x: x * 2
f = lambda x: x * 3

# Calling Functions
print(f(g(10)))
```

Dynamic Functions

- Use function objects to select / define functions at runtime

```
import sys

if sys.platform == 'win32':
    cp = win32_copy
else:
    cp = unix_copy

cp('foo', 'bar')
```

Variable Scope

```
actions = {}

for idx, act in enumerate(['one', 'two', 'three', 'four']):
    actions[act] = lambda: print(idx)

actions['one']()
```

- What is printed? Why? How would you fix it?

Variable Scope

- Variables are scoped in function
- Python2 offers a “global” keyword to bind to a global variable
- Python3 offers a “nonlocal” keyword to bind to an outer scope variable

Closures

```
def counter():
    count = [0]
    def f():
        count[0] += 1
        print(count[0])
    return f

c = counter()
c()
c()
c()
```

```
# Python3

def counter():
    count = 0
    def f():
        nonlocal count
        count += 1
        print(count)
    return f

c = counter()
c()
c()
```

Variadic Functions

- The star operator allows functions to receive a varying number of arguments
- The double star operator allows functions to receive a varying number of keyword arguments

Variadic Functions

```
def sum_power(exp, *numbers):
    return sum(x**exp for x in numbers)

def sum_squares(*numbers):
    return sum(x*x for x in numbers)

# prints 30: 1 + 4 + 9 + 16
print(sum_squares(1, 2, 3, 4))

# prints 100: 1 + 8 + 27 + 64
print(sum_power(3, 1, 2, 3, 4))
```

Variadic Keywords

```
def sum_words(sentence, **weight):
    """ returns sum of all words in sentence, according to weight
specified """
    return sum(weight.get(word, 0) for word in sentence.split())

# prints 90: 10 + 30 + 50
print(sum_words('I can see the mountains', I=10, can=30, see=50))
```

Keyword Only Arguments

```
# Python3 Only
def my_sum(x, y, *, exp=1):
    return (x ** exp) + (y ** exp)

print(my_sum(10, 20))

print(my_sum(10, 20, exp=2))
```



Q & A

Lab: Functions.

Part 1



Higher Order Functions

Higher Order Functions

- A function can return another function
- Useful for conditional modification of a function

Example: Count

- The following functor receives a function and returns a new version of it that counts how many times it was called
- Client code can use it to modify behavior of any function in a specified way

Example: Count

```
def count(f):
    counter = 0
    def wrapped(*args, **kwargs):
        nonlocal counter
        print(
            "{} called so far {} times."
            "Calling again".format(
                f.__name__, counter)
        )
        counter += 1
    return f(*args, **kwargs)

return wrapped
```

Example: Count

```
def say_hi():
    print("hello world")

say_hi = count(say_hi)

say_hi()
say_hi()
say_hi()
```

Decorators

- Decorator syntax is a shortcut for using higher order functions
- Decorating a function will call the decorator (which itself is a functor) and replace the decorated function with the result

Example: Count

```
@count
def say_hi():
    print("hello world")

say_hi()
say_hi()
say_hi()
```

Multiple Decorators

```
@makebold  
@makeitalic  
def hello():  
    return "hello world"
```



Q & A

Generators

Hello Generators

- A generator is a function that creates a sequence
- Python can automatically “save” the function’s state so sequence generation continues where we left off
- The keyword “yield” defines a generator

Hello Generators

```
def sqr():
    i = 1
    while True:
        yield i * i
        i += 1
```

Using Generators

```
import itertools

for x in itertools.islice(sqr(), 10): print(x)
```



Q & A

Lab: Functions.

Part 2



Exception Handling

Why

- Exceptions decouple error handling from business logic

Living Without Exceptions

```
res = sqrt(9)
if res is not None:
    print res
else:
    print "Error: Negative numbers have no sqrt"
```

Raising Exceptions

```
def sqrt(n):
    if n < 0:
        raise ValueError("sqrt Expected"
                         "parameter >= 0,"
                         "Got: %d" % n)
    return math.sqrt(n)

sqrt(-5)
```

Handling Exceptions

```
try:  
    print sqrt(-1)  
except ValueError as e:  
    print "Error was: %s" % e.message  
except Exception as e:  
    print "Something went wrong. Sorry", e  
finally:  
    print "Thanks for using Python Sqrt Calculator"
```

Guidelines

- Exception classes are organized in a hierarchy
- Handle children before parents

Python Exception Hierarchy

```
BaseException
... Exception
.... StandardError
..... TypeError
..... ImportError
..... ZipImportError
..... EnvironmentError
..... IOError
..... ItimerError
..... OSError
..... EOFError
..... RuntimeError
..... NotImplemented
..... NameError
..... UnboundLocalError
..... AttributeError
..... SyntaxError
..... IndentationError
..... TabError
..... LookupError
..... IndexError
..... KeyError
..... CodecRegistryError
..... ValueError
..... UnicodeError
..... UnicodeEncodeError
..... UnicodeDecodeError
..... UnicodeTranslateError
..... AssertionError
..... ArithmeticError
..... FloatingPointError
..... OverflowError
..... ZeroDivisionError
..... SystemError
..... CodecRegistryError
..... ReferenceError
..... MemoryError
..... BufferError
..... StopIteration
..... Warning
..... UserWarning
..... DeprecationWarning
..... PendingDeprecationWarning
..... SyntaxWarning
..... RuntimeWarning
..... FutureWarning
..... ImportWarning
..... UnicodeWarning
..... BytesWarning
..... _OptionError
... GeneratorExit
... SystemExit
... KeyboardInterrupt
```

Custom Exceptions

```
class InvalidVATError(Exception):
    def __init__(self, msg):
        super(InvalidVATError, self).__init__(msg)
    self.message = msg

class Invoice:
    def __init__(self, vat):
        if vat < 0:
            raise InvalidVATError("VAT must be >= 0."
                                  "Got: %d" % vat)
        self.vat = (1.0 + vat / 100.0)
```

Custom Except Hook

```
def global_handler(exctype, value, tb):
    print('My Error Information')
    print('Type:', exctype)
    print('Value:', value)
    print('Traceback:', tb)
    print('-----')
    original_excepthook(exctype, value, tb)

original_excepthook = sys.excepthook
sys.excepthook = global_handler
```



Q & A

Lab: Exceptions



Collections



map, filter, reduce

- map - (collection) -> (another collection)
- filter - (collection) -> (part of that collection)
- reduce - (collection) -> item

map

```
>>> map(lambda x: x * x, range(5))
[0, 1, 4, 9, 16]

// or if you're on python3
>>> map(lambda x: x * x, range(5))
<map object at 0x102709fd0>

>>> list(map(lambda x: x * x, range(5)))
[0, 1, 4, 9, 16]
```

filter

```
>>> filter(lambda n: re.search(r'(\d)\1', str(n)),  
           range(100))  
  
[11, 22, 33, 44, 55, 66, 77, 88, 99]
```

reduce

```
>>> import operator  
>>> reduce(operator.add, range(10))  
45
```

List Comprehension

- A shorthand syntax for map and filter

```
l = [f(x) for x in list if p(x)]
```

List Comprehension

```
>>> [ x*x  
      for x in range(100)  
      if re.search(r'(\d)\1', str(x)) ]  
  
[121, 484, 1089, 1936, 3025, 4356, 5929, 7744, 9801]
```

Creating Generators

- List comprehensions can create generators (same as functions that yield)

```
>>> g = (x*x  
           for x in range(100)  
           if re.search(r'(\d)\1', str(x)))  
  
>>> next(g)  
121  
>>> next(g)  
484
```

Dictionary Comprehension

```
>>> { k : chr(k) for k in range(65, 122) }
```

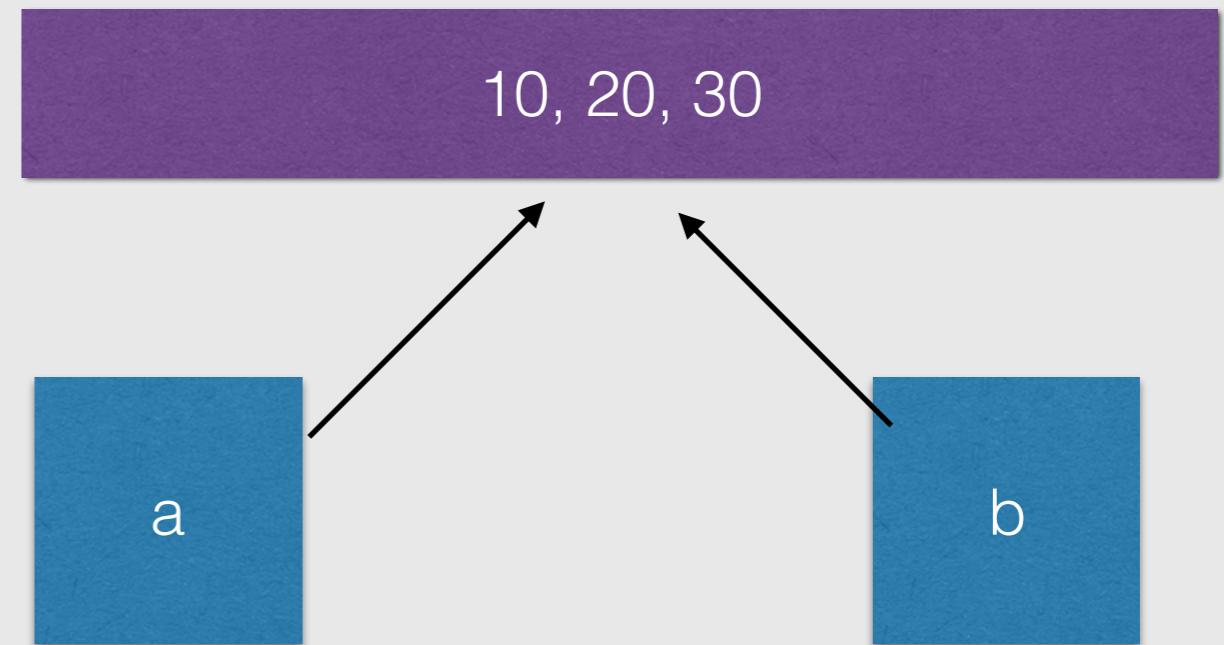
```
{65: 'A', 66: 'B', 67: 'C', 68: 'D', 69: 'E', 70: 'F', 71: 'G', 72: 'H', 73: 'I', 74: 'J', 75: 'K', 76: 'L', 77: 'M', 78: 'N', 79: 'O', 80: 'P', 81: 'Q', 82: 'R', 83: 'S', 84: 'T', 85: 'U', 86: 'V', 87: 'W', 88: 'X', 89: 'Y', 90: 'Z', 91: '[', 92: '\\", 93: ']', 94: '^', 95: '_', 96: '`', 97: 'a', 98: 'b', 99: 'c', 100: 'd', 101: 'e', 102: 'f', 103: 'g', 104: 'h', 105: 'i', 106: 'j', 107: 'k', 108: 'l', 109: 'm', 110: 'n', 111: 'o', 112: 'p', 113: 'q', 114: 'r', 115: 's', 116: 't', 117: 'u', 118: 'v', 119: 'w', 120: 'x', 121: 'y'}
```

Memory Consideration

- Multiple collections can refer to the same item
- When changing that item, all collections that have it are affected

Memory Consideration

```
>>> a = [ 10, 20, 30 ]  
>>> b = a  
>>> a[ 0 ] = 50  
>>> b  
[ 50, 20, 30 ]
```



Copying Lists

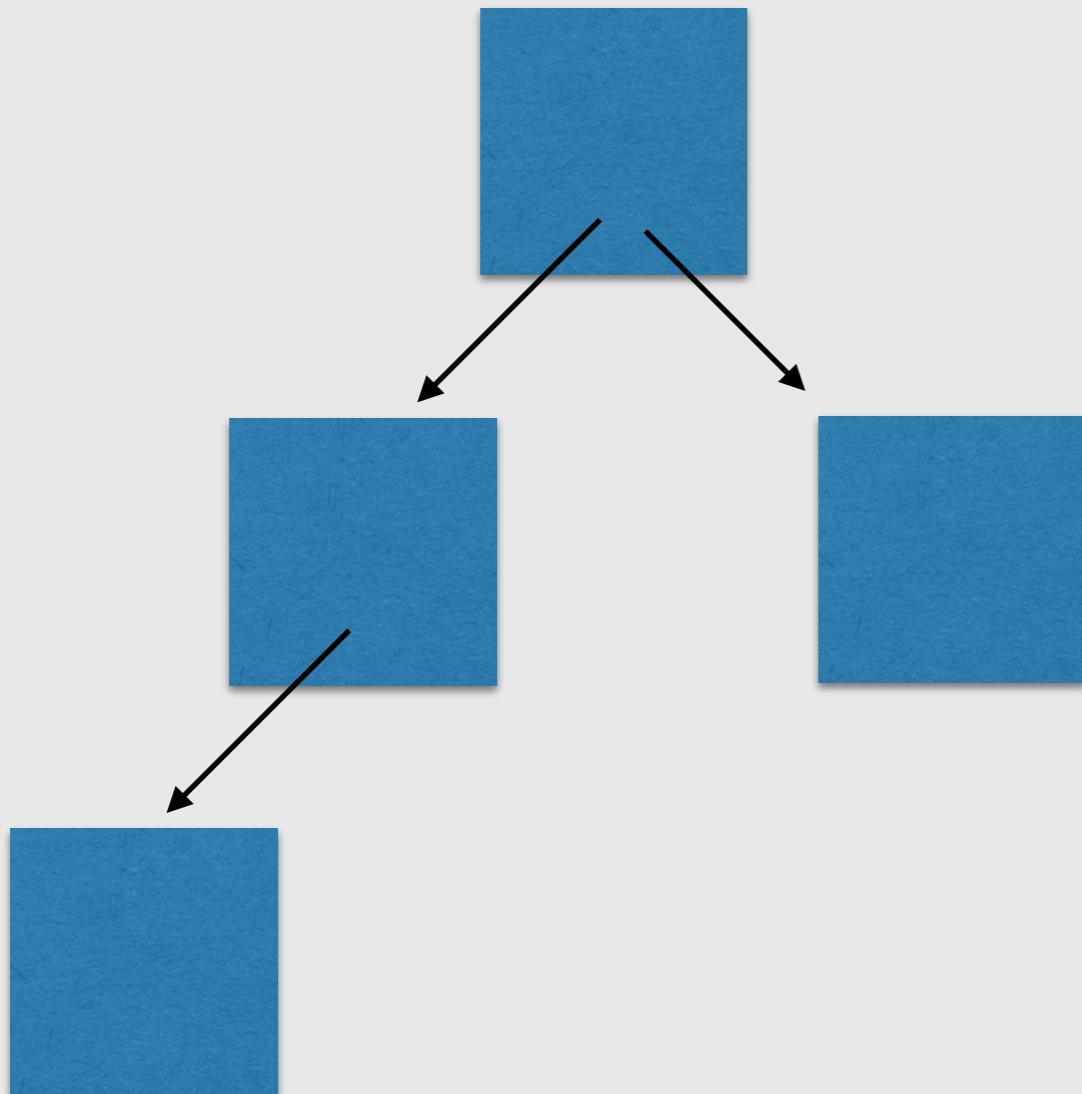
```
>>> a = [10, 20, 30]

# copy a list by slicing the original one
>>> b1 = a[:]

# use list() function to create a new list
>>> b2 = list(a)

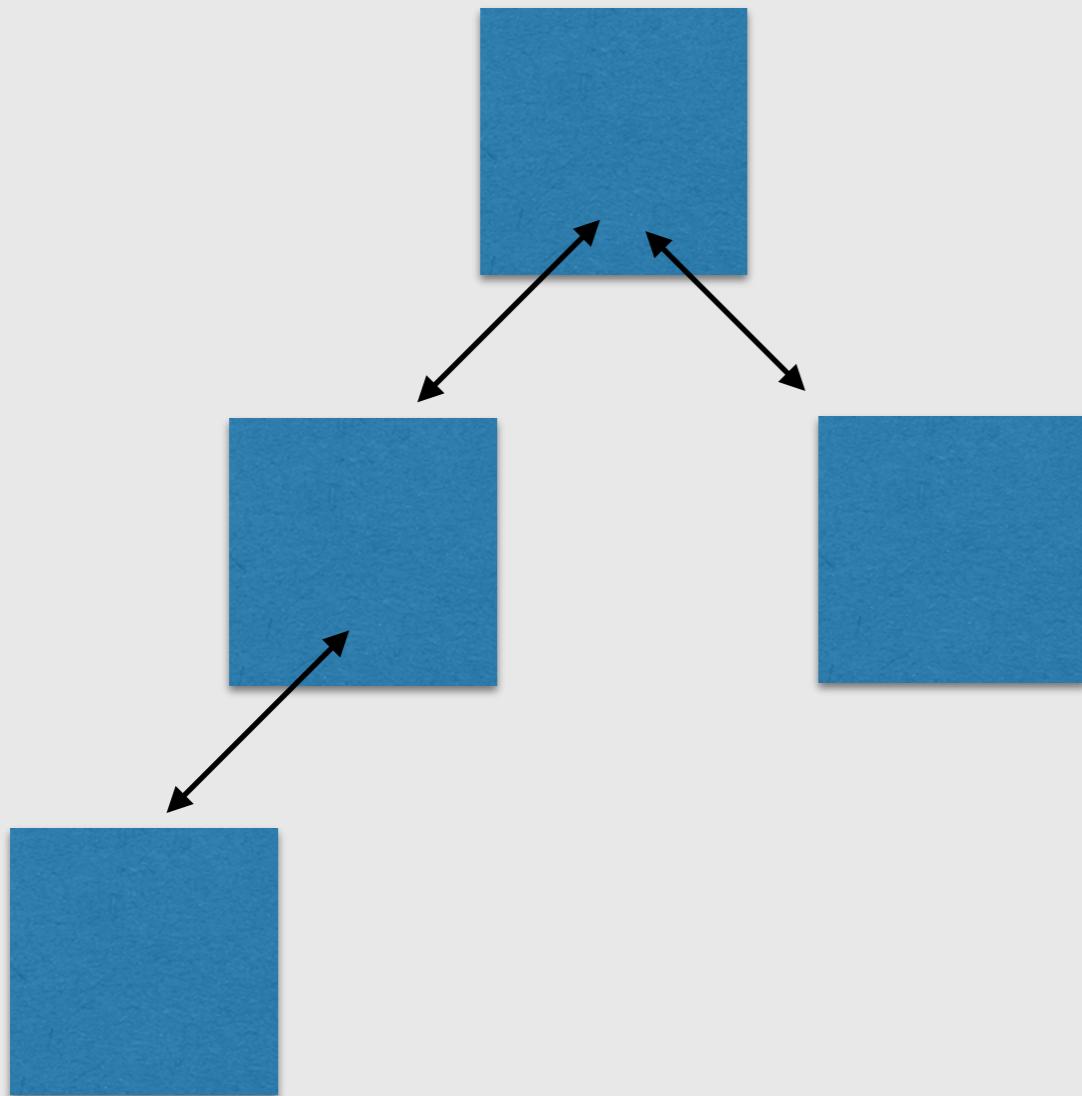
# use copy.copy to create a shallow copy of list
>>> import copy
>>> b3 = copy.copy(a)
>>> b4 = copy.deepcopy(a)
```

Weak vs. Strong ref



- When parent node is deleted, all tree is deleted

Weak vs. Strong ref



- But what happens when we add a pointer back to parent?

Weak vs. Strong ref

- Tree will not be deleted because each node still has other “things” referencing it
- Mainly, the tree keeps itself alive

Code

```
class Node:  
    def __init__(self, value, parent=None, left=None, right=None):  
        self.value = value  
        self.parent = parent  
        self.left = left  
        self.right = right  
  
    def __del__(self):  
        print('Deleting node: {}'.format(self.value))  
  
    def add(self, node, where):  
        node.parent = self  
        if where == 'l':  
            self.left = node  
        else:  
            self.right = node
```

Fixing With Weakref

```
def add(self, node, where):
    node.parent = weakref.proxy(self)
    if where == 'l':
        self.left = node
    else:
        self.right = node
```



Q & A

Lab: Collections



Polymorphism



Encapsulation

Classes In Python

- Integrated into language constructs
- Extend language with a new vocabulary

Classes In Python

```
import sys

with open('/etc/passwd') as f:
    for line in f: print(line)
```

- Which classes take part in the above code?
- What python constructs are in use?

Classes Create Persistent Scoping

- Classes have members, which means they store data persistently between method calls
- This works much like closures in functions

Class Syntax

```
class LightBulb:

    def __init__(self):
        self.is_on = False

    def turn_on(self):
        if self.is_on: return

        print "Lights On"
        self.is_on = True

    def turn_off(self):
        if not self.is_on: return

        print "Lights Off"
        self.is_on = False
```

Method Arguments

```
class Invoice:  
    def __init__(self, vat):  
        self.vat = (1.0 + vat/100.0)  
  
    def print_item(self, name, price):  
        print("Item: %s, Price With VAT: %g" %  
              (name, price * self.vat ))  
  
i = Invoice(18)  
  
# prints: Item: Veggy Burger, Price With VAT: 56.64  
i.print_item("Veggy Burger", 48)
```

Class Variables

```
class Invoice:  
    vat = 1.17  
  
    def print_item(self, name, price):  
        print("Item: {}, Price With VAT: {}".format(  
            name, price * Invoice.vat))  
  
i = Invoice()  
j = Invoice()  
  
i.print_item("Veggy Burger", 48)  
j.print_item("Tofu", 20)  
  
Invoice.vat = 1.16  
i.print_item("Veggy Burger", 48)  
j.print_item("Tofu", 20)
```

Class Methods

```
class Invoice:  
    vats = {  
        "Israel" : 1.18,  
        "Italy" : 1.22,  
        "Japan" : 1.08,  
        "Jordan" : 1.16,  
        "Thailand" : 0.07,  
    }  
  
    @classmethod  
    def vat_by_country(cls, country):  
        return Invoice.vats[country]  
  
print(Invoice.vat_by_country("Israel"))  
print(Invoice.vat_by_country("Thailand"))
```

Properties

- Define properties as member variables on self
- If/when you'll need to add logic, change them to properties with `@property` decorator

Properties

```
class Product:
    def __init__(self, *, name, price):
        self.name = name
        self._price = price

    @property
    def price(self):
        return self._price * 1.17

    @price.setter
    def price(self, value):
        self._price = value
```

Slots

- Slots restrict the names you can use for object attributes
- assign a list of names to class `__slots__` field
- Requires new-style classes

Slots

```
class Critter(object):
    __slots__ = ('name', 'friends')

c = Critter()

# works
c.name = 'dude'

# crash - can't assign to likes
c.likes = 'food'
```



Q & A

Lab: Class Syntax



Class Docstring

- Classes have docstrings (just like functions)
- Use pydoc or help() to read them

Class Docstring

```
class Artist:  
    """ An Artist data object that maintains a song list """  
    def __init__(self, name):  
        """ An Artist constructor. Takes name as str """  
        self.name = name  
        self.songs = []  
  
    def create_song(self, name):  
        """ Create a new song for this artist. Takes song name as str """  
        self.songs.append(Song(self, name))
```

Python/Classes Integration

- Many python contracts will call methods on objects implicitly
- These methods are usually called “magic methods” and have 2 underscores on each side
- Full list:
<https://www.cmi.ac.in/~madhavan/courses/prog2-2012/docs/diveintopython3/special-method-names.html>

Printing

- The method `__repr__` decides how an object is printed
- The method `__str__` decides how an object is translated to string

Printing

```
from __future__ import print_function

class Artist:
    def __repr__(self):
        return "__repr__: I am an artist"

    def __str__(self):
        return "__str__: I am an artist"

a = Artist()
print(a)
print(repr(a))
```

Iteration

- An iterator iterates over a collection in a for loop
- Provides:
 - `next` - get the next item (Python 2)
 - `__next__` - get the next item (Python 3)
 - raise `StopIteration` when no more items are left
- Special method `__iter__` should return an iterator

With

- A with block has `__enter__` and `__exit__` parts
- It is used to handle exceptions (as an alternative to try/except)

With

```
from __future__ import print_function

class Artist:
    def __enter__(self):
        print("Entering with block")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("bye bye")

with Artist() as a:
    pass
```

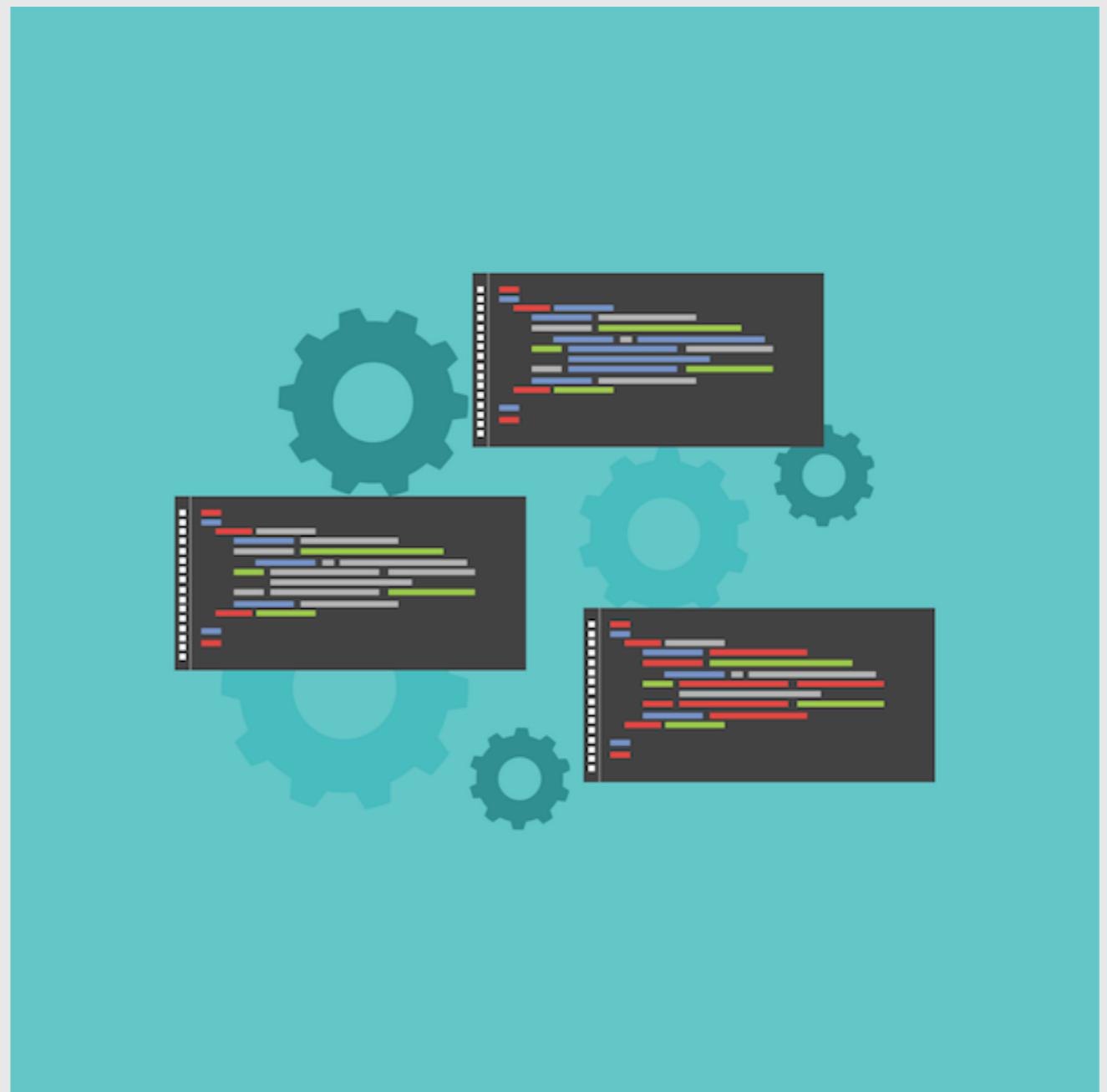


Q & A

Lab: Magic Methods



Multiple Classes



Alternatives

- Message Passing
- Composition
- Inheritance

Messages

- Two independent classes can send messages to each other to achieve common goal

```
p = Product()  
c = CreditCard()  
  
p.buy(c)
```

Composition

```
class ShoppingCart:  
    def checkout(self, creditcard):  
        for p in self.products:  
            p.buy(creditcard)
```

- Defined by a “has-a” relationship between classes
- Sending a message to ShoppingCart causes it to send messages to all its products

Inheritance

```
class Freebie(Product):
    def buy(creditcard):
        # ask user for their
        # phone number instead of
        # taking money
```

- Defined by a “is-a” relationship between classes
- Allows changing a little bit of functionality without rewriting the whole thing

Use Cases

- Use **independent classes** and send messages between them whenever you can
- Use **composition** to build complex relationships between classes
- Use **inheritance** for highly coupled code

Example: Automatic Delegation

- In python it's easy to set up automatic delegation for has-a relationship
- Use the suggested “delegate” function

Example: Automatic Delegation

```
def delegate(obj, to, *methods):
    for method_name in methods:
        setattr(obj, method_name, getattr(to, method_name))
```

```
class Item():
    def __init__(self):
        self.invoice = Invoice()
        delegate(self,
                 self.invoice,
                 'price_with_vat',
                 'print_vat')
```

Example: CountList

- Write a List class that accepts a type and allows adding new items of that type
- Extend that class to remember how many “add” actions were performed
- Use both delegation and inheritance



Q & A

Lab: Multiple Classes



Descriptors

Why

- Use descriptors to share logic between properties

Example: Counter

```
class T(object):
    c = Counter()
    d = Counter()

t = T()

# 1
print(t.c)
# 2
print(t.c)

# 1
print(t.d)
```

Counter Code

```
class Counter(object):
    def __init__(self, val = 0):
        self.val = val

    def __get__(self, obj, cls):
        self.val += 1
        return self.val
```

Descriptor Protocol

- Assign descriptor to class variable
- A descriptor provides:
 - `__get__(self, obj, cls)`
 - `__set__(self, obj, value)`
 - `__delete__(self, obj)`

Alias Descriptor

```
class Alias(object):
    def __init__(self, nm):
        self.nm = nm

    def __get__(self, obj, cls):
        return getattr(obj, self.nm)

    def __set__(self, obj, value):
        setattr(obj, self.nm, value)

    def __delete__(self, obj):
        delattr(obj, self.nm)
```

Usage

```
class T(object):
    bar = Alias('foo')

    def __init__(self):
        self.foo = 10
```



Q & A

Meta-Classes

What

- Each “thing” has a type
- $\text{type}([1,2,3])$ ==> <type ‘list’>
- $\text{type}(\text{type}([1,2,3]))$ ==> <type ‘type’>
- $\text{type}(\text{type}(\text{type}([1,2,3])))$ ==> <type ‘type’>

What

- A meta class is the type of your class
- Default is named “type”

Why

- Run code when class is defined
- Run code when class is called
- Automate descriptor creation

How

```
from __future__ import print_function

class MyMeta(type):
    def __init__(self, name, parents, props):
        super(type, self).__init__(self)
        print(name)
        print(parents)
        print(props)

class T(object):
    __metaclass__ = MyMeta

print('--- the end')
```

How

- Meta classes extend “type”
- Declare meta class with `__metaclass__`
- Meta classes are inherited
- Meta class code is called after class dictionary is created

Automatic Class Registry

```
class Registry(type):
    classes = {}

    def __init__(cls, name, parents, class_props):
        super(type, cls).__init__(cls)
        Registry.classes[name] = cls

    @classmethod
    def new_instance(cls, name, *args, **kwargs):
        return Registry.classes[name](*args, **kwargs)
```

Using It

```
class T(object):
    __metaclass__ = Registry
    def __init__(self): print('I am a T')

class F(object):
    __metaclass__ = Registry
    def __init__(self): print('I am an F')

Registry.new_instance('T')
```

Calling Metaclasses

- Object creation = calling the meta class
- By implementing `__call__` we can control how objects are created

MLogger

```
from __future__ import print_function

class MLogger(type):
    def __call__(self, *args, **kwargs):
        print('Creating new object: {}'.format(self))
        return type.__call__(self, *args, **kwargs)

class T(object):
    __metaclass__ = MLogger
    def __init__(self, val):
        self.val = val

t = T(10)
r = T(20)
```

Singleton

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(
                Singleton, cls).__call__(*args, **kwargs)
    return cls._instances[cls]

class T(object):
    __metaclass__ = Singleton
    def __init__(self):
        print('New T is being created...')
```

Automatic Descriptors

- The last parameter to `__init__` is the class dictionary
- We can use it to create descriptors automatically

Automatic Descriptors

```
class CounterMaker(type):
    def __init__(cls, name, parents, props):
        super(type, cls).__init__(cls)
        print(props)
        for c in props.get('counters', []):
            setattr(cls, c, Counter())
```



Q & A

Modules



201

Hello Modules

- As a program grows we split it to files
- An external .py file is called a **package**
- The keyword **import** searches and loads a package

Search Path

- Folder of the importer
- PYTHONPATH
- Default system paths
- The following prints python's search path

```
python -c 'import sys; print sys.path'
```

Alert

- Python is not great when module names conflict with system names
- Demo: <https://codp.in/c/python-import-demo-1493302469>

Namespaces

```
# file main.py
import funcs

# prints 30
print(funcs.sum(10, 20))

# prints 6
print(funcs.mul(2, 3))
```

```
# file funcs.py
def sum(x, y):
    return x + y

def mul(x, y):
    return x * y
```

Project Structure

```
/project
    /lib
        funcs.py
```

```
/tests
    test_funcs.py
```

```
main.py
```

Testing

- To run your tests:
`python -m unittest discover`
- Write all tests in tests/ folder in files that start with “test_”
- Example: project with tests:
https://github.com/tocodeil/python-course-examples/tree/master/26_dirstructure

Writing Tests

- A test case is a class extending unittest.TestCase

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

unittest.main()
```

setup/teardown

```
class TestFile(unittest.TestCase):

    def setUp(self):
        self.dir = tempfile.mkdtemp()
        self.name = self.dir + "/file1"
        self.file = File(self.name)

    def tearDown(self):
        shutil.rmtree(self.dir)
```

Mocking Functions

```
import unittest
from mock import patch
import os

class TestIt(unittest.TestCase):
    @patch('os.getpid', return_value=10)
    def test_pid(self, fake_getpid):
        self.assertEqual(os.getpid(), 10)
        fake_getpid.assert_called_once()

unittest.main()
```

Mocking stdout

```
from mock import patch
from StringIO import StringIO

class TestEx2(unittest.TestCase):
    @patch('sys.stdout', new_callable=StringIO)
    @patch('random.randint', return_value=10)
    def test_sum_7_random_integers(self, rand_spy, out_spy):
        execfile('02.py')
        self.assertIn('70', out_spy.getvalue(),
                     'Expected to find 70 but got: %s' %
out_spy.getvalue())
```

MagicMock object

```
import json
import unittest
from mock import patch
import os

class TestIt(unittest.TestCase):
    @patch('json.load')
    def test_pid(self, fake_loads):
        fake_loads.return_value = { 'foo': 10, 'bar': 20 }
        data = json.load()

        self.assertEqual(data['foo'], 10)

unittest.main()
```

Test Assertions

```
assertIs(a, b)  
assert IsNot(a, b)  
assertIsNone(x)  
assert IsNotNone(x)  
assertIn(item, collection)  
assertNotIn(item, collection)  
assertIsInstance(a, b)  
assertNotIsInstance(a, b)
```

Mock Assertions

```
thing.assert_called()
thing.assert_called_once()
thing.assert_called_with(1, 2, 3, test='wow')
thing.assert_called_once_with('other', bar='values')
thing.assert_any_call(1, 2, arg='thing')
thing.assert_not_called()
thing.call_count
thing.call_args_list
```



Q & A

Lab: Modules



Using External Modules

Intro

- Python Package Index - is the official site for sharing python code
- Anyone can upload a distribution
- Install and use with “**pip**”

Setting Up Virtual Env

```
# linux / mac
$ pip install -U pip setuptools
$ pip install virtualenv

# windows
python -m pip install -U pip setuptools
python -m pip install virtualenv
```

Creating Our venv

```
$ mkdir -p ~/venv/work  
$ virtualenv ~/venv/work
```

```
# or for a specific python version  
$ virtualenv -p /usr/bin/python2.7 ~/venv/work
```

Enter / Leave

```
# Enter a virtual env  
$ source ~/venv/work/activate
```

```
# Leave a virtual env  
$ deactivate
```

Demo

- Create a virtual env
- Install IsraelIDChecker

Requirements.txt file

- Every project that uses external modules need to list them so others can easily install
- Current standard is a requirements.txt file
- Create with:

```
$ pip freeze > requirements.txt
```

- Install from requirements.txt:

```
$ pip install -r requirements.txt
```

requirements.txt

```
Cython>=0.19.2
numpy>=1.7.1
scipy>=0.13.2
scikit-image>=0.9.3
matplotlib>=1.3.1
ipython>=3.0.0
h5py==2.2.0
leveldb>=0.191
networkx>=1.8.1
```



Q & A

Multiprocessing

Parallel Python

multiprocessing

threading

thread

Things to do in parallel

- CPU bound tasks
- IO bound tasks

thread

- low level OS code
- changed to `_thread` in python3

threading

- Thread
- Lock
- Condition
- Semaphore
- Event

Hello Threads

```
from threading import Thread
import thread

def go():
    for i in range(10):
        print("Hi! I'm thread {id} counting {i}".format(
            id=thread.get_ident(),
            i=i))

for _ in range(4):
    t = Thread(target=go)
    t.start()

threads = [Thread(target=go) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
```

Threads

- Threads run at the same time
- `t.start()` starts thread t and calls its target
- `t.join()` waits for thread t to finish
- Python is completely thread-safe due to GIL

multiprocessing

- Process
- Queue
- Pool
- Pipe
- All sync primitives from threading

What's to like

- Used to perform multiple tasks in parallel both by splitting to processes AND threads
- Easy to switch between the two due to a unified interface

What's not to like

- CAUTION: processes need to pickle data
- Not everything can be shared

Hello Multiprocessing

```
from multiprocessing import Pool
from math import sqrt

def isprime(x):
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
    return x

if __name__ == '__main__':
    p = Pool(5)
    print([n for n in
          p.map(isprime, range(2, 500000))
          if n])
```

Hello Multiprocessing

- Multiple processes: 0m2.036s
- Multiple threads: 0m6.076s
- Single thread: 0m3.607s

Takeaways

- CPU bound tasks work best when split to processes
- Pool.map is an effective way to split work

Threads & Network

- Network and blocking IO code can benefit from threads
- We can use the same multiprocessing API

Threads & Network

```
from multiprocessing.dummy import Pool

urls = [
    "https://www.dropbox.com/s/ihrsnxnvix96ep/407H.jpg?dl=1",
    "https://www.dropbox.com/s/64n7yfi8i08jwdr/406H.jpg?dl=1",
    "https://www.dropbox.com/s/uk6kv7suuo7zazv/403H.jpg?dl=1",
    "https://www.dropbox.com/s/i129qs82zcu4l7r/409H.jpg?dl=1",
]

p = Pool(4)
p.map(download, urls)
```

Threads & Network

- 4 Threads: 1m4.473s
- 4 Processes: 1m4.604s
- Conclusion: IO bound tasks don't need true processes

Using Queue

- Pool is useful when you know the task in advance
- Queue is useful for dynamic tasks
- Demo: find in files

Using Queues

Fill Queue with file names

Get items from queue

Search in file content

Thread Synchronization: Locks, Semaphores, Conditions

Locks

- Protect critical sections that can not be executed in parallel (from multiple threads)

```
from __future__ import with_statement  
  
with lock:  
    ... access shared resource
```

Semaphores

- Protect limited resources than can only be “used” by N threads at the same time

```
semaphore = threading.BoundedSemaphore(5)
semaphore.acquire()
... access the shared resource
semaphore.release()
```

Event

- An imaginary “flag” that is used to notify that some resource is ready

```
event = threading.Event()

# a client thread can wait for the flag to be set
event.wait()

# a server thread can set or reset it
event.set()
event.clear()
```

Condition

- An imaginary “flash” that is used to indicate some temporary condition was met (i.e. new item is ready)

Condition

```
# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()
```

Takeaways

- Use multiprocessing module to increase performance on relevant CPU bound tasks
- Use multiprocessing module to increase performance on relevant disk bound tasks
- Consider async (twisted) for network related code



Q & A

Lab: multiprocessing



NumPy Introduction

Hello NumPy

```
import numpy as np

ar = np.array([
    [ '#', ' ', '#'],
    [ '#', ' ', '#'],
    [ ' ', '#', ' '],
])

print(np.flipud(ar))
```

Why NumPy

- Numpy vs. Matlab
- Numpy vs. Regular Python

Numpy Vs. Python

```
import numpy as np
import random

# Create a vector of 1 million
# random values
m = np.random.rand(1_000_000)

# Create a list of 1 million
# random values
x = [random.randint(0,100) for _ in range(1_000_000)]

# Sum the vector using numpy
# time: 414 µs ± 19.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
np.sum(m)

# Sum the list using python
# time: 7.9 ms ± 272 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
sum(x)
```

Building Arrays

```
import numpy as np

# A random vector
m1 = np.random.rand(1_000_000)

# A random 2d matrix size 10x10
# (rows, cols)
m2 = np.random.rand(10, 10)

# A random 5d array
m3 = np.random.rand(5, 4, 3, 5, 5)
```

Building Arrays

```
import numpy as np

# Building arrays from values
m = np.array([
    [10, 20, 30],
    [20, 30, 40],
    [30, 40, 50],
])
```

Building Arrays

```
import numpy as np
from io import BytesIO

# Building arrays from external data
data = b"""
10,20,30
20,30,40
40,50,60
"""

# pass a file, a stream or a file name
m = np.genfromtxt(BytesIO(data), delimiter=',')
print(m)
```

Building Arrays

```
import numpy as np

# Identity matrix
i = np.identity(5)

# A 4x4 zeros matrix
m0 = np.zeros(16).reshape(4, 4)

# A 4x4 matrix with the value 5
m5 = (np.ones(16) * 5).reshape(4, 4)
```

Slicing

- Numpy allows us to access:
 - Specific elements
 - Specific rows or column
 - Ranges of rows and columns

Slicing

```
import numpy as np

m = np.random.rand(3, 3)

# Prints the top left element
print(m[0,0])

# Modify the top left element
m[0, 0] = 5
```

Slicing

```
import numpy as np

m = np.random.rand(3, 3)

# Prints the first row
print(m[0])
print(m[0, :])

# Modify the top row
m[0] = [10, 20, 30]
```

Slicing

```
import numpy as np  
  
m = np.random.rand(9, 9)  
  
# Print the upper right part  
print(m[0:3, 6:])
```

Boolean Masks

```
arr = np.array([10, 20, 30, 31, 32])
arr > 20
# Returns: array([False, False,  True,  True,  True],
dtype=bool)

arr[arr > 20]
# Returns:  array([30, 31, 32])
```

Data Types

- Array's data types selects the type of its elements

```
import numpy as np

m = np.arange(9).reshape(3,3)

# int64
print(m.dtype)

m = np.random.rand(3,3)

# float64
print(m.dtype)
```

Setting Data Type

```
import numpy as np

m = np.arange(9, dtype='float64').reshape(3, 3)

# float64
print(m.dtype)
```

Available Data Types

- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- float16, float32, float64, float96, float128
- bytes_, unicode_, void
- Full list:
<https://docs.scipy.org/doc/numpy/reference/arrays.scalars.html#arrays-scalars-built-in>

Arrays Operations

- NumPy has overloaded operators between arrays
- In case an operand isn't an array, it'll be *broadcasted* to one if possible
- We can define custom vectorised functions

Arrays Operations

- Multiply element-by-element. Return a new array with the same shape

```
import numpy as np

m = np.arange(9).reshape(3,3)
n = np.arange(9).reshape(3,3)

print(m * n)
```

Arrays Operations

- Works for boolean conditions too. Returns an array of booleans

```
import numpy as np

m = np.arange(9).reshape(3,3)
n = np.arange(9).reshape(3,3)
m[0,0] = 5

print(m == n)
```

Broadcasting

- Operations between arrays of different shapes will try to broadcast the operands to match shape
- Vectors can “grow” in only one dimension
- Scalars will “grow” as much as needed

Broadcasting

```
import numpy as np  
  
m = np.arange(9).reshape(3,3)  
  
# Add 5 to each element  
m + 5
```

Broadcasting

```
import numpy as np

m = np.arange(9).reshape(3, 3)

# Add [0, 1, 2] to each row
print(m + np.array([0, 1, 2]))

# Add [0, 1, 2] to each column
print(m + np.array([[0], [1], [2]]))
```

Data Type

```
import numpy as np

x = np.arange(9, dtype='int64').reshape(3,3)
y = np.arange(9, dtype='int64').reshape(3,3)

# Works: x is now an array of floats
x = x * 10.0
print(x)

# Fails: int * float = float,
# but y is an array of ints
y *= 10.0
```

Universal Functions

- A “ufunc” is a function that operates on one or more arrays.
- NumPy has tons of them listed here:
<https://docs.scipy.org/doc/numpy/reference/ufuncs.html>

Unary Functions

```
import numpy as np

m = np.random.rand(5,5)

# Round all values
print(np.rint(m))

# Square all values
print(np.square(m))
```

Binary Functions

```
import numpy as np

# Randomize a 5x5 matrix values 0..100
m = np.floor(np.random.rand(5,5) * 100)

# Print remainder when dividing each element by 3
print(np.remainder(m, 3))

# Turn all values > 50 into 50
print(np.fmin(m, 50))

# Combine ufunc with slicing
m[np.greater(m, 20)] = 0
```

Custom ufuncs

```
import numpy as np

def _add_if_lessthan_10(a, b):
    if a + b < 10:
        return a + b
    else:
        return a

add_if_lessthan_10 = np.vectorize(_add_if_lessthan_10)

m = np.arange(9).reshape(3, 3)
print(add_if_lessthan_10(m, 5))
```



Q & A

Lab: NumPy



Consuming Web APIs with Python and requests

Requests Module

- Requests is an elegant and simple HTTP library for Python, built for human beings.

Install

- pip install requests
- python -m pip install requests

Hello World

```
import requests

response = requests.get('https://swapi.co/api/people/1/')
response.raise_for_status()
data = response.json()

print(data['name'])
```

Print Facebook Likes

```
import requests

access_token = '....'

response = requests.get('https://graph.facebook.com/me/likes',
                        params={ 'access_token': access_token })
response.raise_for_status()
data = response.json()

print([like['name'] for like in data['data']])
```

POSTing Data

```
import requests

# send data to
# http://postb.in/b/hzYyocpa

# Send parameters as query string
response = requests.post('http://postb.in/hzYyocpa',
                         params={ 'name': 'ynon' })
response.raise_for_status()
print(response.content)
```

POSTing Data

```
import requests

# send data to
# http://postb.in/b/hzYyocpa

# Send parameters as POST body
response = requests.post('http://postb.in/hzYyocpa',
                          data={ 'name': 'ynon' })
response.raise_for_status()
print(response.content)
```

Downloading Files

```
import requests

r = requests.get(
    'https://cdn.trendhunterstatic.com/phpthumbnails/
     90/90449/90449_1_800.jpeg')
filename = 'funnycar.jpg'

with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size=512):
        fd.write(chunk)
```

Sessions & Cookies

- Session maintains the same TCP connection and uses keep-alive to improve performance
- Session maintains cookies between requests
- Can set common data to all requests

Using Sessions

```
s = requests.Session()
r = s.get('https://swapi.co/api/people/1/')
r.raise_for_status()
data = r.json()

for url in data['films']:
    r = s.get(url)
    r.raise_for_status()
    film_data = r.json()
    print(film_data['title'])
```

Common Params

```
import requests
access_token = '...'

photos_count = 0
next_url = 'https://graph.facebook.com/25750098595/photos'
s = requests.Session()
s.params = { 'access_token': access_token }

while True:
    response = s.get(next_url)
    response.raise_for_status()
    data = response.json()

    photos_count += len(data[ 'data' ])
    try:
        next_url = data[ 'paging' ][ 'next' ]
    except KeyError:
        break

print(photos_count)
```

Requests Lab

- Find your facebook album with the maximum number of photos
- Try to speed up the code by using multiple threads
- Try to speed up the code by using multiple processes
- **Recommendation:** Use one session per thread



Q & A

Thanks For Listening

- Ynon Perek
- ynon@tocode.co.il
- ynon@ynonperek.com
- www.tocode.co.il/blog