Python Libraries

DS 8015

Outline

Imports

TERMINOLOGY

- Module: Smallest unit of code reusability

  File containing Python definitions and statements

- Package: Logical collection of modules

  Often bundles large products and broad functionality

- Standard Library: Collection of packages and modules

  Distributed with Python by default

- Script: Any Python code invoked as an executable

  Usually from the command line

IMPORT FROM A MODULE

```
# Import a module
import math
math.sqrt(16) # => 4

# Import symbols from a module into the local namespace
from math import ceil, floor
ceil(3.7) # => 4.0
floor(3.7) # => 3.0

# Bind a module symbols to a new local symbol
from some_module import long_symbol_name as short_name

# Any python file (including your own) can be a module
from my_script import my_function, my_variable
```

IMPORTING FROM PACKAGES

```
sound/
├── __init__.py
├── effects/
│   ├── __init__.py
│   ├── echo.py
│   ├── reverse.py
│   └── surround.py
├── filters/
│   ├── __init__.py
│   ├── equalizer.py
│   ├── karaoke.py
│   └── vocoder.py
└── formats/
    ├── __init__.py
    ├── aiffread.py
    ├── aiffwrite.py
    ├── auread.py
    ├── auwrite.py
    ├── wavread.py
    └── wavwrite.py
```

```
# Packages give structure to modules
import sound.effects.echo
sound.effects.echo.echofilter(input, output)

from sound.effects import echo
echo.echofilter(input, output, delay=0.7,
 atten=4)

from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)

# A namespace, in a sense...
#__init__.py can distinguish packages from
# normal directories
```

Package Import Rules

```
# The item can be a submodule (or subpackage) of package
from package import item

# All but the last must be packages
import item.subitem.subsubitem
```

- Import Conventions:
  - Imports go at the top of the file after header comment.
    ⇒ Why? Clear dependencies, avoid conditional imports
  - Prefer import ... to from ... import ...
    ⇒ Why? Explicit namespaces avoid name conflicts
  - Avoid from ... import *
    ⇒ Why? Unclear what is being imported, strange behavior

Executing Modules as Scripts

Refresher: Running Modules as Scripts

```
# We can run a module (demo.py) as a script
$ python3 demo.py # Doing so sets __name__ = '__main__'
<output>
# We can even jump into the interpreter after we're done
$ python3 -i demo.py
<output>
>>> # Access to top-level symbols
```

The Standard Library

FRAMING

- We've moved beyond the Python language (syntax, spec)
- "Python" is a "batteries-included" distribution
- Many powerful tools are already implemented for us in The Standard Library [Click me!]

- Goal is to become aware of Python's numerous utilities.

STANDARD LIBRARY TOUR

- collections - container datatypes: namedtuple / defaultdict / Counter
- re - regular expressions: patterns for strings
- itertools - iterators for efficient looping
- json - encode and decode structured JSON data
- random - generate pseudo-random numbers
- sys - interact with system
- pathlib - intelligent filesystem navigation
- subprocess/shlex - spawn processes
- debugging - breakpoint(), pprint, timeit
- cute modules - turtle, unicodedata, this, antigravity
- builtins - any, all, int, hex, bin, ord, chr, round, max, min, sum, pow, divmod

COLLECTIONS.NAMEDTUPLE - 1

```
# use to create tuple subclasses with NAMED FIELDS
import collections
Point = collections.namedtuple('Point', ['x', 'y'])

p = Point(11, 22) # positional arguments...
q = Point(x=11, y=22) # or keyword arguments (or both!)

# Fields are accessible by name! "Readability counts."
-p.x, 2 * p.y # => -11, 44

# readable __repr__ with a name=value style
print(p) # Point(x=11, y=22)

# Subscriptable, like regular tuples
p[0] * p[1] # => 242
# Unpackable, like regular tuples
x, y = p # x == 11, y == 22
# Usually don't need to unpack if attributes have names
math.hypot(p.x - other.x, p.y - other.y) # raises error
```

COLLECTIONS.NAMEDTUPLE - 2

```
# Can you guess the context of this code?

p = (170, 0.1, 0.6)
if p[1] >= 0.5:
    print("Whew, that is bright!")
if p[2] >= 0.5:
    print("Wow, that is light!")

# BAD!
```

COLLECTIONS.NAMEDTUPLE - 3

```
# alternative:

Color = collections.namedtuple("Color",
    ["hue", "saturation", "luminosity"])

pixel = Color(170, 0.1, 0.6)
if pixel.saturation >= 0.5:
    print("Whew, that is bright!")
if pixel.luminosity >= 0.5:
    print("Wow, that is light!")

# GOOD!
```

COLLECTIONS.DEFAULTDICT - 1

```
# dict subclass with factory function for missing values

# Have:
input_data = [('yellow', 1), ('blue', 2),
    ('yellow', 3), ('blue', 4), ('red', 1)]

# Want:
output = {'blue': [2, 4], 'red': [1], 'yellow': [1, 3]}
```

COLLECTIONS.DEFAULTDICT - 2

```
input_data = [('yellow', 1), ('blue', 2),
    ('yellow', 3), ('blue', 4), ('red', 1)]

# One approach
output = {}
for k, v in input_data:
    if k not in output:
        output[k] = []
    output[k].append(v)
print(output)
# => {'blue': [2, 4], 'red': [1], 'yellow': [1, 3]}
```

COLLECTIONS.DEFAULTDICT - 3

```
# A better approach

#accepts one argument - a zero-argument
# factory function to supply missing keys
output = collections.defaultdict(lambda: list())
for k, v in input_data:
    output[k].append(v)
      # When key is missing, go to the factory

print(output)
#=> defaultdict(<function <lambda> at 0x........>,
# {'red': [1], 'yellow': [1, 3], 'blue': [2, 4]})
```

COLLECTIONS.DEFAULTDICT - 4

```
# Zero-Argument Callable

# defaultdict with default value []
collections.defaultdict(lambda: list())

# equivalent to
collections.defaultdict(list)

# defaultdict with default value 0
collections.defaultdict(lambda: 0)

# equivalent to
collections.defaultdict(int)
```

COLLECTIONS.DEFAULTDICT - 5

```
# Have: s = 'mississippi'
# Want: d = {'i': 4, 'p': 2, 'm': 1, 's': 4}

s = 'mississippi'
d = collections.defaultdict(int) # or... lambda: 0

for letter in s:
    d[letter] += 1
print(d)
#=> defaultdict(<class 'int'>,
# {'i': 4, 'p': 2, 'm': 1, 's': 4})
```

COLLECTIONS.COUNTER - 1

```
# dict subclass for counting hashable objects

# Have: s = 'mississippi'
# Want: [('s', 4), ('m', 1), ('i', 4), ('p', 2)]
s = 'mississippi'

count = collections.Counter(s)

print(count)
# => Counter({'i': 4, 'm': 1, 'p': 2, 's': 4})

print(list(count.items()))
# => [('s', 4), ('m', 1), ('i', 4), ('p', 2)]
```

COLLECTIONS.COUNTER - 2

```
# Tally occurrences of words in a list
colors = ['red', 'blue', 'red', 'green', 'blue']

# One approach
counter = collections.Counter()
for color in colors:
    counter[color] += 1
print(counter)
# Counter({'blue': 2, 'green': 1, 'red': 2})

# A better approach
counter = collections.Counter(colors)
print(counter)
# Counter({'blue': 2, 'green': 1, 'red': 2})
```

COLLECTIONS.COUNTER - 3

```
# Get most common elements!
collections.Counter('abracadabra').most_common(3)
# => [('a', 5), ('b', 2), ('r', 2)]

# Supports basic arithmetic
collections.Counter('which')
+ collections.Counter('witch')
# => Counter({'c': 2, 'h': 3, 'i': 2, 't': 1, 'w': 2})

collections.Counter('abracadabra')
- collections.Counter('alakazam')
# => Counter({'a': 1, 'b': 2, 'c': 1, 'd': 1, 'r': 2})
```

RE - 1

```
# Regular expression operations
# "regular expression" == "search pattern" for strings

#Search for pattern match anywhere in string;
# return None if not found
# \w matches word characters
# + is for one or more occurrences of preceding expression
import re
m = re.search(r"(\w+) (\w+)", "Isaac Newton, Physicist")
m.group(0) # "Isaac Newton" - the entire match
m.group(1) # "Isaac" - first parenthesized subgroup
m.group(2) # "Newton" - second parenthesized subgroup

#Match pattern against start of string;
# return None if not found
m = re.match(r"(?P<fname>\w+) (?P<lname>\w+)", "Jeff Fox")
m.group('fname') # => 'Jeff'
m.group('lname') # => 'Fox'
```

RE - 2

```
# Substitute occurrences of one pattern with another
re.sub(r'@\w+\.com', '@stanford.edu',
    'sam@go.com poohbear@bears.com')
# => sam@stanford.edu poohbear@stanford.edu

# compile pattern for fast operations
# [a-z]: Match any lowercase
# [0-9]: Match any digit
# {3}: match exactly three digits
pattern = re.compile(r'[a-z]+[0-9]{3}')
# pattern is first argument
match = re.search(pattern, '@@@abc123')
match.span() # (3, 9)
#span() Return a tuple containing the (start, end)
# positions of the match
```

RE - 3

```
"""
Write a regular expression to match a phone number like
650 867-5309
Hint: \d captures [0-9], i.e. any digit
Hint: \d{3} captures 3 consecutive digits
"""
def is_phone(num):
    return bool(re.match('\d{3} \d{3}-\d{4}', num))

def get_area_code(num):
    m = re.match('(?P<areacode>\d{3}) \d{3}-\d{4}', num)
    if not m:
        return None
    return m.group('areacode')

is_phone("650 867-5309") # => True
is_phone("650.867.5309") # => False
get_area_code("650 867-5309") # => '650'

# Done? Use named groups to return the area code
```

RE - 4

```
# Find the three most common words in Hamlet
with open('hamlet.txt') as f:
    words = re.findall(r'\w+', f.read().lower())

collections.Counter(words).most_common(3)
# => [('the', 1091), ('and', 969), ('to', 767)]

with open('hamlet.txt') as f:
    words = re.findall(r'\w{5}', f.read().lower())

collections.Counter(words).most_common(3)
# => [('queen', 121), ('hamle', 117), ('there', 116)]
```

ITERTOOLS - 1

```
# iterators for efficient looping
# COMBINATORICS
import itertools

def view(it): print(*map(''.join, it))

view(itertools.product('ABCD', 'EFGH'))
# => AE AF AG AH BE BF BG BH CE CF CG CH DE DF DG DH
view(itertools.product('ABCD', repeat=2))
# => AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD

view(itertools.permutations('ABCD', 2))
# => AB AC AD BA BC BD CA CB CD DA DB DC

view(itertools.combinations('ABCD', 2))
# => AB AC AD BC BD CD

view(itertools.combinations_with_replacement('ABCD', 2))
# => AA AB AC AD BB BC BD CC CD DD
```

ITERTOOLS - 2

```
# INFINITE ITERATORS

# start, [step] -> start, start + step, ...
itertools.count(10) # -> 10, 11, 12, 13, 14, ...

# Cycle through elements of an iterable
itertools.cycle('ABC') # -> 'A', 'B', 'C', 'A', ...

# Repeat a single element over and over.
itertools.repeat(10) # -> 10, 10, 10, 10, ...
```

JSON

```
# JSON encoder and decoder

import json
squares = {1:1, 2:4, 3:9, 4:16}
# Serialize to/from string
output = json.dumps(squares) # output == "{1:1, 2:4, 3:9, 4:16}"
json.loads(output) # => {1:1, 2:4, 3:9, 4:16}

# Serialize to/from file
with open('tmp.json', 'w') as outfile:
    json.dump(squares, outfile)

with open('tmp.json', 'r') as infile:
    input = json.load(infile)

# All variants support useful keyword arguments
json.dumps(data, indent=4, sort_keys=True, separators=(',', ': '))
```

RANDOM - 1

```
# Generate pseudo-random numbers

import random

# Random float x with 0.0 <= x < 1.0
random.random() # => 0.3744488717564646

# Random float x, 1.0 <= x < 10.0
random.uniform(1, 10) # => 1.1800146073117523

# Random integer from 1 to 6 (inclusive)
random.randint(1, 6) # => 4 (https://xkcd.com/221/)

# Random integer from 0 to 9 (inclusive)
random.randrange(10) # => 7

# Random even integer from 0 to 100 (inclusive)
random.randrange(0, 101, 2) # => 26
```

RANDOM - 2

```
# Choose a single element
random.choice('abcdefghij') # => 'c'

items = [1, 2, 3, 4, 5, 6, 7]
random.shuffle(items)
items # => [7, 3, 2, 5, 6, 4, 1]

# k samples without replacement
random.sample(range(5), k=3) # => [3, 1, 4]

# Sample from statistical distributions (others exist)
random.normalvariate(mu=0, sigma=3) # => 2.37378057827
```

BUILTIN FUNCTIONS - 1

```
any([True, True, False]) # => True
all([True, True, False]) # => False

int('45') # => 45
int('0x2a', 16) # => 42
int('1011', 2) # => 11
hex(42) # => '0x2a'
bin(42) # => '0b101010'

ord('a') # => 97
chr(97) # => 'a'

round(123.45, 1) # => 123.4
round(123.45, -2) # => 100
```

BUILTIN FUNCTIONS - 2

```
max(2, 3) # => 3
max([0, 4, 1]) # => 4
min(['apple', 'banana', 'pear'], key=len) # => 'pear'

sum([3, 5, 7]) # => 15

pow(3, 5) # => 243 (= 3 ** 5)
pow(3, 5, 10) # => 3 (= (3 ** 5) % 10, efficiently)

quotient, remainder = divmod(10, 6)
# quotient, remainder => (1, 4)

# Flatten a list of lists (slower than itertools.chain
sum([[3, 5], [1, 7], [4]], []) # => [3, 5, 1, 7, 4]
```

Other Modules - 1

- string - Common string operations
- struct - Interpret bytes as packed binary data
- datetime - Basic date and time types
- fractions - Rational numbers
- statistics - Mathematical statistics functions
- operator - Standard operators as functions
- pickle - Python object serialization
- csv - CSV File Reading and Writing
- os - Miscellaneous operating system interfaces

Other Modules - 2

- time - Time access and conversions
- argparse - Parser for command-line options, arguments and sub-commands
- logging - Logging facility for Python
- threading - Thread-based parallelism
- multiprocessing - Process-based parallelism
- socket - Low-level networking interface
- asyncio - Asynchronous I/O, event loop, coroutines and tasks

Third-party packages

NumPy - 1

- *N*-dimensional array object
- Sophisticated functions
- Capabilities
  - Linear algebra
  - Fourier transform
  - Random sampling
- NumPy Docs [Click me]

NumPy - 2

```
import numpy as np
a = np.arange(15).reshape(3, 5)
print(a)
# array([[ 0, 1, 2, 3, 4],
# [ 5, 6, 7, 8, 9],
# [10, 11, 12, 13, 14]])

a.shape # => (3, 5)
a.ndim # => 2
a.dtype.name # => 'int64'
type(a) # => numpy.ndarray

print(a[:, 1]) # => array([ 1, 6, 11])
```

NumPy - 3

```
a = np.array([3, 4, 5])
a + 4 # => array([7, 8, 9])
a * 1.5 # => array([ 4.5, 6. , 7.5])

b = np.array([4, -1, 0])
np.dot(a, b) # => 8 (= 3 * 4 + 4 * -1 + 5 * 0)
a.sum() # => 12

# 100 interpolated numbers between 0 and 6.28
space = np.linspace(0, 2 * np.pi, 100)
sinusoid = np.sin(space)
# trigonometry - pi corresponds to 180 degrees
```

SciPy

- Everything you need for mathematics, science, and engineering.
- SciPy Docs [Click me]

```
import numpy as np
from scipy import linalg

# Invert a matrix
A = np.array([[1, 2], [3, 4]])
print(linalg.inv(A))
# array([[-2. , 1. ],
# [ 1.5, -0.5]])
```

MATPLOTLIB + PYPLOT

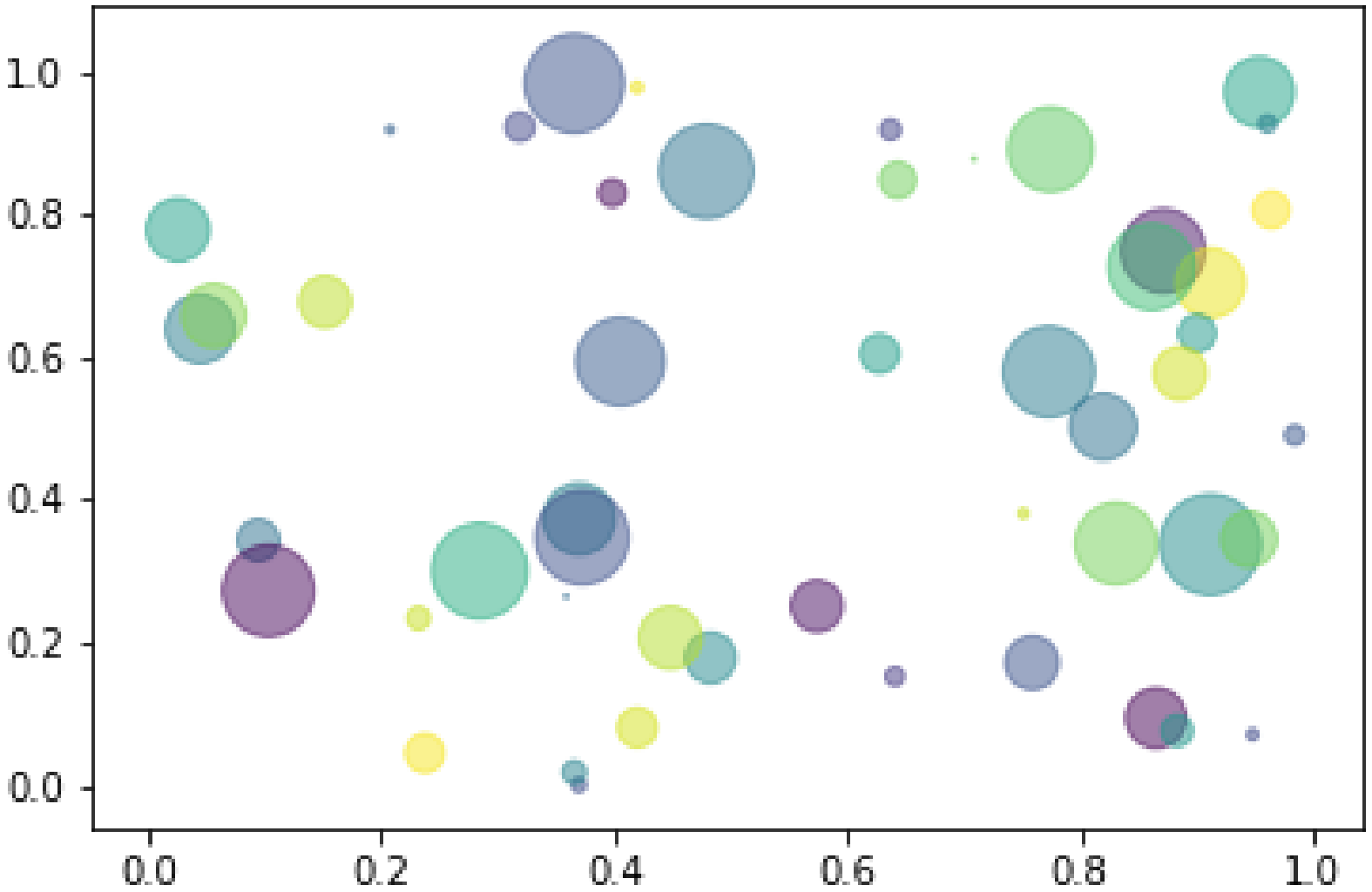
- Python 2D Plotting Library
- Think MATLAB + Python
- Examples [Click me]
- Gallery [Click me]

```
import numpy as np
import matplotlib.pyplot as plt

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
plt.savefig('scatter.png') # comment out plt.show()
```

Scatter Plot



SOME OTHER IMPORTANT LIBRARIES & PACKAGES

- tensorflow (MI)
- keras (high-level API)
- scikit-learn (ML)
- nltk (Natural Language)
- pytorch (Deep Learning)
- pandas (Vectorized data manipulation)