The Python Standard Library

May 9, 2017

Overview

Imports Redux

Python's Standard Library!

The Big Picture

Behind Us - The Python Language

Week 1 Python Fundamentals

Week 2 Data Structures

Week 3 Functions

Week 4 Functional Programming

Week 5 Object-Oriented Python

The Road Ahead - Python Tools

Week 6 Standard Library

Week 7 Third-Party Tools

Week 8 Ecosystem

Week 9 Advanced Topics

Week 10 Projects!

Before We Begin: Semantics

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

Importing from Modules

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

Packages

Packages give structure to modules

```
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
```

```
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 distinguishes 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
```

Good Python: Import Conventions

Import Conventions

Imports go at the top of the file after header comment

Why? Clear dependencies, avoid conditional imports

Prefer `import ...` instead of `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
```

Aside: Finding Modules

Searching "Algorithm"

if builtin module exists:

    load builtin module

else:

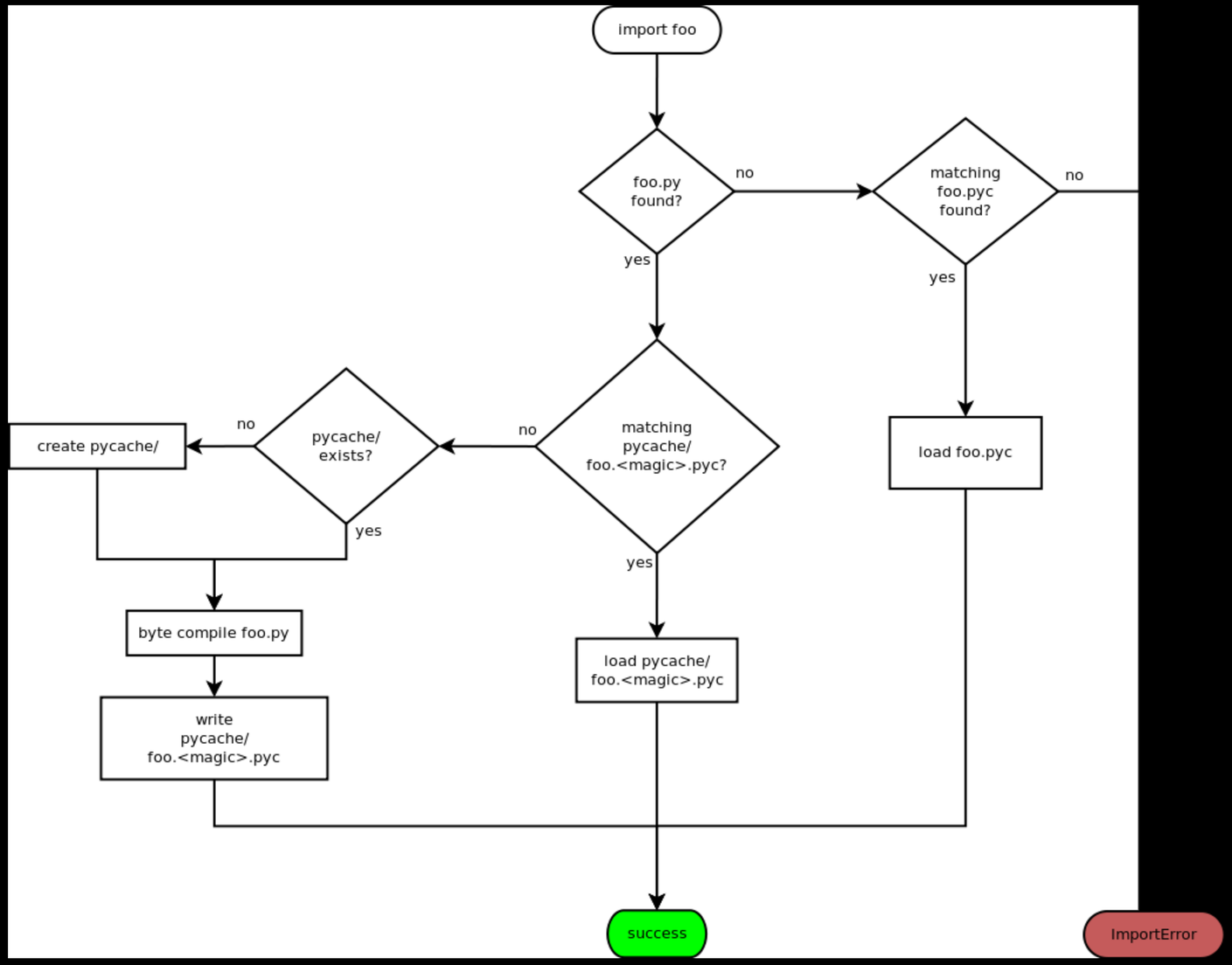
    look for builtin module in the current directory of script

    look through PYTHONPATH

    look in installation default

load if found, else raise ImportError

Searching with Caches



Taken straight from PEP 3147

What's up with `*.pyc`?

CPython will cache the byte-compiled modules (.pyc)

- Cached .pyc files live in `__pycache__/module.vers.pyc`

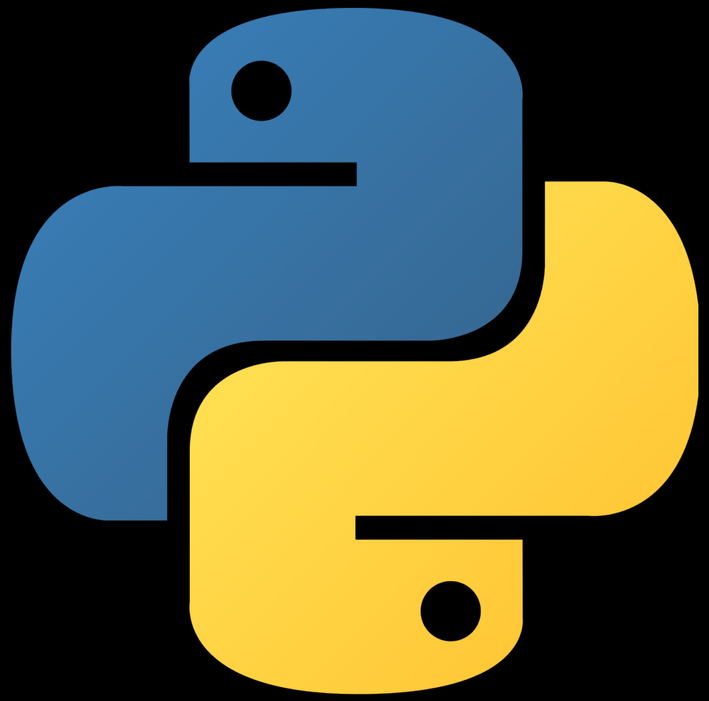
Automatically recompiled when source code is newer

Doesn't change runtime speed, only loading speed

More info here

Time-Out for Announcements

Logistics



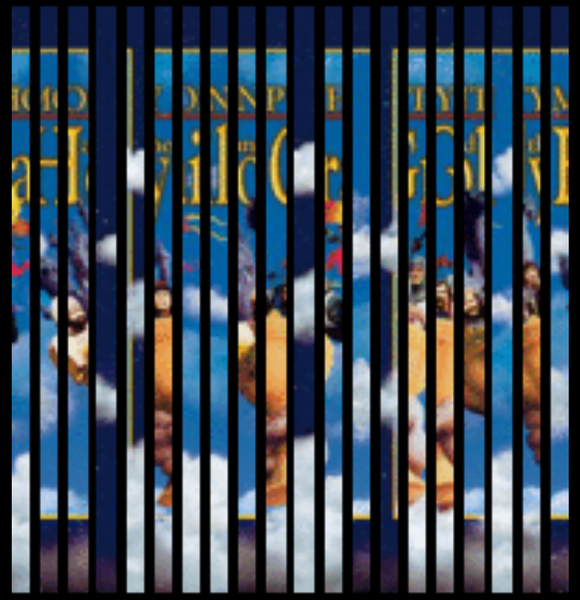
Assignment 1 Grades

Assignment 2 OH

- 7PM-9PM Tuesday
- 7PM-9PM Wednesday

Tressider by Starbucks

Logistics



Start Early!

GForm for Final Submission

+1 Late Day!

Back to Python!

The Standard Library

Overview

Behind: Python syntax and philosophy

"Python" is a "batteries-included" distribution

Many powerful tools are already implemented in the:

Standard Library

Click me!

Disclaimer

Goal: Awareness of Python's numerous utilities

- Roughly sorted by importance and relevance to CS41

Slides are written as reference materials - a "pitch" for each

Ask questions! Run examples!

Assume all necessary `import`s have been executed

Bread and Butter

`collections`
container datatypes

`collections.namedtuple`

create tuple subclasses with named fields

collections.namedtuple

```
Point = collections.namedtuple('Point', ['x', 'y'])

p = Point(11, y=22)  # positional or keyword arguments

# 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)
```

collections.namedtuple

```
Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(11, 22)

# Subscriptable, like regular tuples
p[0] * p[1]  # => 242

# Unpack, 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)
```

Good Python Style:
Use `namedtuple`

collections.namedtuple

```
# 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

```
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!")
```

`collections.defaultdict`

dict subclass with factory function
for missing values

collections.defaultdict

```
# 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

```
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

```
input_data = [...]

# A better approach
output = collections.defaultdict(lambda: list())
for k, v in input_data:
    output[k].append(v)


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

accepts one argument - a zero-argument factory function to supply missing keys

When key is missing, go to the factory

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)
```

Your Turn

```
# 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`

dict subclass for counting hashable objects

collections.Counter

```
# 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

```
# 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

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

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

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

re

Regular expression operations

"regular expression" == "search pattern" for strings

`re` — Regular expression operations

```
# Search for pattern match anywhere in string; return None if not found
m = re.search(r"(\w+) (\w+)", "Physicist Isaac Newton")
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+)", "Malcolm Reynolds")
m.group('fname')  # => 'Malcolm'
m.group('lname')  # => 'Reynolds'
```

`re` — Regular expression operations

```
# 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

pattern = re.compile(r'[a-z]+[0-9]{3}')  # compile pattern for fast ops
match = re.search(pattern, '@@@abc123')  # pattern is first argument
match.span()  # (3, 9)
```

Your Turn

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

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

Your Turn

```
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')
```

collections.Counter and re

```
# 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)]
```

`itertools`

iterators for efficient looping

Combinatorics

```
def view(it): print(*[''.join(els) for els in 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
```

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

`json` — JSON encoder and decoder

Similar module for CSV

```
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`

Generate pseudo-random numbers

`random` — Generate pseudo-random numbers

```
# Random float x with 0.0 <= x < 1.0
random.random()  # => 0.37444887175646646

# 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` — Generate pseudo-random numbers

```
# 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)  # => 2.373780578271
```

`sys`

System-specific
parameters and functions

`sys` — System-specific parameters and functions

```
# Open file objects for standard input, error, output
sys.stdin ('r') / sys.stderr ('w') / sys.stdout ('w')
sys.stdin.readline()
sys.stderr.write('hello world\n')
sys.stdout.flush()

# Raise SystemExit
sys.exit(arg)
```

One more thing...

Refresher: Running Modules as Scripts

```
# We can run a module (demo.py) as a script
$ python3 demo.py  # Doing so sets __name__ = '__main__'

# We can even jump into the interpreter after we're done
$ python3 -i demo.py

# What if we want to do something like...
$ python3 -i demo.py <arguments>
```

`sys.argv` to the rescue!

```
# File: demo.py
if __name__ == '__main__':
    import sys
    print(sys.argv)
```

```
$ python3 demo.py 1 2 3
['demo.py', '1', '2', '3']
$ python3 subdir/../demo.py foo
['subdir/../demo.py', 'foo']
```

For more advanced command line tools, use argparse (if needed, cmd and getopt)

System Interaction

`pathlib` — Object-oriented filesystem paths

```
p = pathlib.Path('/etc')
q = p / 'ssh'  # Overloaded __div__ method
q  # => PosixPath('/etc/ssh')

q.exists()  # => True
q.is_dir()  # => True

# Print all python files somewhere in the current dir
p = pathlib.Path.cwd()  # Current working directory
for f in p.glob('**/*.py'):
    print(f)
```

subprocess and shlex

```
subprocess.call(["ls", "-l"])  # => 0

# Automatically authenticate to Myth servers
command = "kinit name@myth.stanford.edu --keytab=/etc/some-keytab"
args = shlex.split(command)  # args = ["kinit", ... ]
subprocess.call(args)  # => 0

# For more complex needs, use Popen
# Emulate 'ps aux | grep Spotify'
sp_ps = subprocess.Popen(["ps", "aux"], stdout=subprocess.PIPE)
sp_grep = subprocess.Popen(["grep", "Spotify"], stdin=sp_ps.stdout)
```

Debugging Tools

`pprint` — data pretty printer

```
# Some horrendous data structure
ugly = {
    'data': {
        'after': 't3_3q8aog',
        'before': None,
        'kind': 'pagination',
        'children': [{'a':1}, {'a':2}, {'b':1}, {}],
        'uuid': '40b6f818'
    }
}
ugly['recursive'] = ugly  # Contains recursive reference
```

`pprint` — data pretty printer

```
print(ugly)
# {'data': {'before': None, 'kind': 'pagination',
'uuid': '40b6f818', 'after': 't3_3q8aog', 'children':
[{'a': 1}, {'a': 2}, {'b': 1}, {}]}, 'recursive': {...}}

pprint.pprint(ugly, width=56, depth=2)
# {'data': {'after': 't3_3q8aog',
#           'before': None,
#           'children': [...],
#           'kind': 'pagination',
#           'uuid': '40b6f818'},
#  'recursive': <Recursion on dict with id=4372885384>}
```

`timeit` - time short snippets

```
# Command Line Interface
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop

# Python Interface
import timeit
timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
# => 0.3018611848820001
timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
# => 0.2727368790656328
timeit.timeit('"-".join(map(str, range(100)))', number=10000)
# => 0.23702679807320237
```

"Cute" Modules

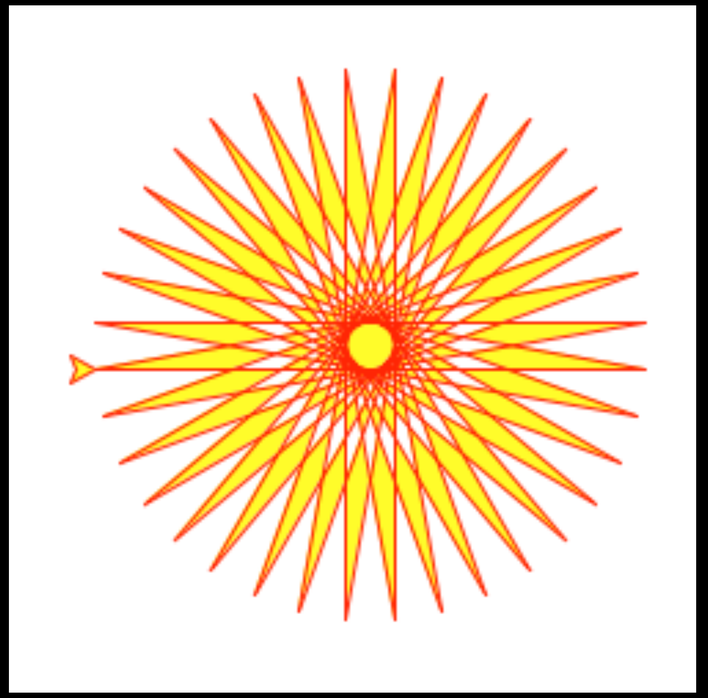
I couldn't resist!

`turtle` — Turtle graphics

```
turtle.color('red', 'yellow')
turtle.begin_fill()

while True:
    turtle.forward(200)
    turtle.left(170)
    if abs(turtle.pos()) < 1:
        break

turtle.end_fill()
turtle.done()
```



`unicodedata` — Unicode Database

```
unicodedata.lookup('SLICE OF PIZZA')
# => '🍕'


unicodedata.name('👌')
# => 'OK HAND SIGN'

unicodedata.numeric('¾')
# => 0.75
```

`this` — Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
...
```

antigravity

```
# "Python is pretty cool! It seems we can do anything."
# "Anything? Do you really mean anything?"
# "I wonder if..."
>>> import antigravity
```

Builtin Functions

Common One-Liners

```
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
```

Common One-Liners

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

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

Modules that you should know exist

Other Modules

6.1. string — Common string operations

7.1. struct — Interpret bytes as packed binary data

8.1. datetime — Basic date and time types

9.5. fractions — Rational numbers

9.7. statistics — Mathematical statistics functions

10.3. operator — Standard operators as functions

12.1. pickle — Python object serialization

14.1. csv — CSV File Reading and Writing

16.1. os — Miscellaneous operating system interfaces

Other Modules

16.3. time — Time access and conversions

16.4. argparse — Parser for command-line options, arguments and sub-commands

16.6. logging — Logging facility for Python

17.1. threading — Thread-based parallelism

17.2. multiprocessing — Process-based parallelism

18.1. socket — Low-level networking interface

18.5. asyncio — Asynchronous I/O, event loop, coroutines and tasks

Other Modules

18.8. signal — Set handlers for asynchronous events

26.3. unittest — Unit testing framework

26.6. 2to3 - Automated Python 2 to 3 code translation

27.3. pdb — The Python Debugger

27.6. trace — Trace or track Python statement execution

29.12. inspect — Inspect live objects

Module Questions?

Summary

Python is "batteries-included"

If you need it, it's probably been implemented for you

Just the tip of the iceberg!

Next Time

Lab

Explore the Standard Library

Read documentation!

Practice with these modules

Next Week

3rd Party Tools

Work Time: Holy Grail!