

Warmup

Think about what the following code will print out!

```
item_list = [x for x in range(10)]  
  
for i, item in enumerate(item_list):  
    print(item)  
    item_list.remove(item)  
  
print(item_list)
```

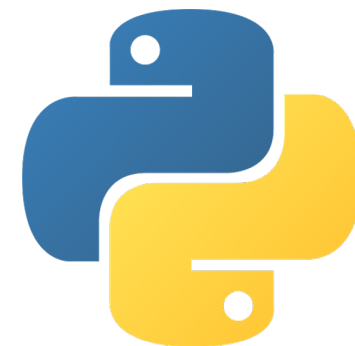
Standard Library and Third-Party Tools

February 19, 2020

Week 7 of CS 41

Today in CS41: Standard Library!

- Modules and Packages
- `pdb` – Python's Debugger!
- `collections` – Specialized Containers for Data!
- `itertools` – Better Iterators!
- `functools` – Functions on Callable Objects!
- `re` – Regular Expressions, Pattern Matching!
- `pickle` – Object Serialization!
- `pathlib` – Object-Oriented Filesystem!
- `threading` – Thread-Based Parallelism!



Modules and Packages

A quick crash course!

Glossary

- **Module:** smallest Python unit of code reusability.
 - File containing Python definitions and statements.
- **Package:** logical collection of modules.
 - E.g. Numpy aggregates hundreds of different modules to provide broad functionality.
- **Standard Library:** collection of packages and modules, distributed with Python by default.

Review - Import from a Module

Import a module

```
import math
```

```
math.sqrt(16)
```

Import functions from a module into the local namespace

```
from math import ceil, floor
```

```
ceil(3.7)
```

```
floor(3.7)
```

Bind a module symbols to a new local symbol

```
from math import degrees as deg
```

```
deg(45.7)
```

Review - Every Python File Is a Module

- Including your own! (There's nothing “official” about module status).
- This allows you to organize larger projects into collections of many files; you can then reference functions in other files using the import statement.

Importing from Packages

File structure of the "sound" package

sound/

- `__init__.py`
- `formats/`
 - `__init__.py`
 - `wavread.py`
 - `wavwrite.py`
 - `aiffwrite.py`
 - `auwrite.py`
 - `...`
- `effects/`
 - `__init__.py`
 - `echo.py`
 - `reverse.py`
 - `...`
- `filters/`
 - `__init__.py`
 - `equalizer.py`
 - `karaoke.py`
 - `...`

*# You can import at various depths of the
package file tree.*

Top-level import - long and unwieldy!

```
import sound
sound.effects.echo.echofilter(input, output)
```

Second-level import

```
from sound.effects import echo
echo.echofilter(input, output)
```

Module-level import

```
from sound.effects.echo import echofilter
echofilter(input, output)
```


Pythonic Import Conventions

- Imports go at the top of a file – this makes clear the dependencies and avoids conditional imports.
- Prefer `import ...` to `from ... import ...`, as this avoids possible name conflicts.
- Try and avoid `from ... import *`. This imports everything without prefixing it to the module's name.
 - Since you probably don't know *everything* that's being imported, so this can lead to name conflicts.

The Standard Library

A brief guided tour!

Why does this matter?

Framing

- Python is a “batteries included” distribution that includes many pre-implemented powerful tools.
- **Today’s Goal:** become aware of some of Python’s numerous utilities. This is a *breadth first* lecture!
 - Also a building point for your own future investigations!
- **TL;DR** if you want it, Python’s standard libraries may well have it! (And it’s very likely a third-party package would have it!)

Today's Tour of the Standard Libraries

- `pdb` – Python's Debugger!
- `collections` – Specialized Containers for Data!
- `itertools` – Better Iterators!
- `functools` – Functions on Callable Objects!
- `re` – Regular Expressions, Pattern Matching!
- `pickle` – Object Serialization!
- `pathlib` – Object-Oriented Filesystem!
- `threading` – Thread-Based Parallelism!

The `pdb` Package

Debugging with Python!

Entering pdb

```
# Code  
breakpoint()  
# More Code
```

```
import pdb  
# Code  
pdb.set_trace()  
# More Code
```

Shell-Based Debugger

- Once at a breakpoint, the terminal program will pause execution.
- The terminal will then prompt you with `(Pdb)` rather than with `>>>` or `$`.
 - You can enter commands at this point to step through the program, or to display information about the program and its variables at various times throughout execution.
 - The principles are identical to the Qt Creator debugger which you've likely seen in CS106B, though instead of a GUI, we have a command line interface!
- Many commands have abbreviations – this just means that, for example, the `next` and `n` commands have the same effect.

Moving Around

- Once you've examined information at one breakpoint, you may want to navigate through your program to look further!

(Pdb) <code>continue</code>	<i># Continue execution until the another # breakpoint is reached. Abbreviated c.</i>
(Pdb) <code>next</code>	<i># Execute the next line of the program, then # stop. Abbreviated n.</i>
(Pdb) <code>step</code>	<i># Execute the next line of the program. If that # line contains a function call, step into the # function, then stop. Appreviated s.</i>
(Pdb) <code>return</code>	<i># Execute until the current function returns. # Appreviated r.</i>

Setting Breakpoints Within pdb

- You may want to set additional breakpoints during the debugging process!

```
(Pdb) break # Show all breakpoints. Abbreviated b.

(Pdb) break lineno # Set a new breakpoint at the line lineno.
# Abbreviated b.

(Pdb) break funname # Set a breakpoint at the first line of the
# function funname. Abbreviated b.
```

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's Debugger!
- `collections` – Specialized Containers for Data!
- `itertools` – Better Iterators!
- `functools` – Functions on Callable Objects!
- `re` – Regular Expressions, Pattern Matching!
- `pickle` – Object Serialization!
- `pathlib` – Object-Oriented Filesystem!
- `threading` – Thread-Based Parallelism!

The collections Package

Specialized containers for your data!

collections.namedtuple

```
# Quick - guess what this code refers to!
```

```
p = ("y1WP2aidto.json", "CikNQoaeFM.json")
```

```
# Obviously, we're using a tuple to express travelling east and south  
# from a given node in the maze assignment. Obviously. 😊
```

```
# This one?
```

```
p = ("dGhE2Sitt0.json", "BhK1uFSdmg.json")
```

```
# This time, we're talking about going west and north. Obviously.
```

`collections.namedtuple`

- Tuples can be awkward – elements of a tuple are referenced by index, not by name.
 - Can lead to confusion!

```
# Using namedtuple to make our example better!
```

```
Maze_Point = collections.namedtuple("Maze_Point", ["north", "east", "south", "west"])
```

```
p = Maze_Point(None, "y1WP2aidto.json", "CikNQoaeFM.json", None)
```

```
p.east           #=> "y1WP2aidto.json"
```

```
p.south         #=> "CikNQoaeFM.json"
```

```
# Much better!
```

collections.defaultdict

```
# Use case - data processing!
```

```
# Say we have:
```

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

```
# And let's say we want:
```

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

```
# What's the best way to do this?
```


collections.defaultdict

```
# Here's one way!
```

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

```
d = {}
```

```
for k, v in input_data:
```

```
    # Need to initialize values for keys which aren't in the dictionary
```

```
    if k not in d.keys():
```

```
        d[k] = []
```

```
    d[k].append(v)
```

collections.defaultdict

Here's another way!

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

```
d = {}
```

```
for k, v in input_data:
```

```
    d[k] = d.get(k, []) + [v]
```

`collections.defaultdict`

- A `defaultdict` takes in any function that returns an object (called a "default factory" in this context).
 - If a key is not in the `defaultdict`, rather than throwing an error, the `defaultdict` calls the default factory and returns the value to the caller.

```
# Here's a beautiful way!

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

# Make the default value an empty list - that way we don't need to set
# a value for keys which aren't in the dictionary already.
d = collections.defaultdict(list)
for k, v in input_data:
    d[k].append(v)
```


collections.Counter

Counting occurrences of hashable objects in a collection!

Let's say we have:

```
animals = "
```

And let's say we want:

```
num_each_animal = {'' : 4, '' : 3, '' : 2, '' : 1}
```

collections.Counter does this for us!

```
num_each_animal = dict(collections.Counter(s))
```

collections.Counter returns a Counter object:

collections.Counter(s) = Counter({'': 4, '': 3, '': 2, '': 1})

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's debugger!
- ~~collections~~ – specialized containers for data!
- `itertools` – better iterators!
- `functools` – functions on callable objects!
- `re` – regular expressions, pattern matching!
- `pickle` – object serialization!
- `pathlib` – object-oriented filesystem!
- `threading` – thread-based parallelism!

The `itertools` Package

Better iterators for more creative loops!

itertools.product

- Enables some nifty combinatoric looping over datasets! Iterates over all combinations of elements of each (iterable) parameter.
- Produces `itertools.product` iterable: items can be obtained using a `for` loop or using a call to `next()`.

```
itertools.product("ABCD", "EFGH")  
# Loops over ('A', 'E'), ('A', 'F'), ('A', 'G'), ('A', 'H'), ('B', 'E') ...  
  
itertools.product("ABCD", "EFGH", "IJKL")  
# Loops over ('A', 'E', 'I'), ('A', 'E', 'J'), ('A', 'E', 'K') ...  
  
itertools.product("ABCD", repeat=2)  
# Loops over ('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A') ...  
# Same as itertools.product("ABCD", "ABCD")
```

Cartesian Product Example

- Running a marketing agency
 - Three different advertising channels
 - Ten demographics
 - Five products
- You want to break down purchase rate of each product based on advertising channel and demographic.

```
channels = [...] # iterable of marketing channels
products = [...] # iterable of products
demographics = [...] # iterable of demographics

for segment in itertools.product(channels, products, demographics):
    # Run an analysis on the segment
```


Combinatorics with `itertools`

- `itertools.permutations` and `itertools.combinations` operate in similar fashion.

```
itertools.permutations("ABCD", 2)
# Loops over all 2-length permutations of "ABCD":
# ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'C') ...
# Unlike in itertools.product, order is a factor!

itertools.combinations("ABCD", 2)
# Loops over all 2-length combinations of "ABCD":
# ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')
```

Infinite Iterators in `itertools`

- `itertools` also provides tools to create infinite iterators. Here are some neat examples!

```
itertools.count(start, step)
# Loops over start, start + step, start + 2*(step), start + 3*(step), ...

itertools.cycle("ABC")
# Loops over A, B, C, A, B, C, A, ...

itertools.repeat(32)
# Loops over 32, 32, 32, 32, 32, 32, ...
```

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's debugger!
- ~~collections~~ – specialized containers for data!
- ~~itertools~~ – better iterators!
- **functools** – functions on callable objects!
- **re** – regular expressions, pattern matching!
- **pickle** – object serialization!
- **pathlib** – object-oriented filesystem!
- **threading** – thread-based parallelism!

The `functools` Package

Operations on callable objects!

`functools.total_ordering`

- Recall comparisons from last week's lab!
 - If we're creating an arbitrary class, we need to define comparisons.
 - Why?
 - E.g. Python doesn't know how to compare two `StanfordCSCourse` objects – it contains arbitrary methods and arbitrary attributes!
- Comparisons defined via magic methods: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`.
- If a class contains `__eq__` and one of the other comparison magic methods, decorating the class with `functools.total_ordering` automatically fills in the rest!

functools.total_ordering

- Recall comparisons from last week's lab!
 - If we're creating an arbitrary class, we need to define comparisons.
 - Why?
 - E.g. Python doesn't know how to compare two `StanfordCSCourse` objects – it contains arbitrary methods and arbitrary attributes!
- Comparisons defined via magic methods: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`.
- If a class contains `__eq__` and one of the other comparison magic methods, decorating the class with `functools.total_ordering` automatically fills in the rest!
 - Defines all the relevant comparison functions.

functools.total_ordering

- Here, `functools.total_ordering` automatically implements the other comparison functions we need in this class!

```
@functools.total_ordering
class Student:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __eq__(self, other):
        return ((self.last_name.lower(), self.first_name.lower()) ==
                (other.last_name.lower(), other.first_name.lower()))

    def __lt__(self, other):
        return ((self.last_name.lower(), self.first_name.lower()) <
                (other.last_name.lower(), other.first_name.lower()))
```

Now... why might this syntax look familiar?

functools.wraps

- Let's revisit an old example from lecture: decorators!

```
# Our first decorator
def debug(function):
    def modified_function(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return modified_function

@debug
def foo(a, b, c=1):
    """
    Bar! (Ok in real life this would be an actual docstring).
    """
    return (a + b) * c
```

What is:

foo.__name__ ?

foo.__doc__ ?

functools.wraps

- Let's revisit an old example from lecture: decorators!
- We would expect `foo.__name__` to be "foo", and `foo.__doc__` to be the docstring comment.
- Instead...
 - `foo.__name__` becomes "debug"!
 - `foo.__doc__` becomes ""!
- Recall that the `@debug` syntax is equivalent to:

```
foo = debug(foo)
```
- So the name and docstring of `foo` are overwritten by those of our decorator!

functools.wraps

- Protect against this by using `functools.wraps`!

```
# Our first decorator
def debug(function):
    def modified_function(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return modified_function

@debug
def foo(a, b, c=1):
    """
    Bar! (Ok in real life this would be an actual docstring).
    """
    return (a + b) * c
```

functools.wraps

- Protect against this by using `functools.wraps`!

```
# Our first decorator
def debug(function):
    @functools.wraps(function)
    def modified_function(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return modified_function

@debug
def foo(a, b, c=1):
    """
    Bar! (Ok in real life this would be an actual docstring).
    """
    return (a + b) * c
```

reduce

- Originally part of `functools`, `reduce` is now a builtin! Nevertheless, it's a nifty function to know!
- Given an iterable of values, and a function of two variables, apply the function sequentially across pairs, until it has been reduced to a single value.

```
# Start with a list of values
values = [1, 2, 3, 4, 5, 6, 7]

# Let's use reduce to turn this into the integer: 1234567
# So our function will be lambda x, y: 10*x+y

reduce(lambda x, y: 10*x+y, values)      #=> 1234567
```

reduce

- Let's take a closer look at what we did!

```
# List of values  
values = [1, 2, 3, 4, 5, 6, 7]  
  
# Reducing function  
lambda x, y: 10*x+5
```

[1, 2, 3, 4, 5, 6, 7]

reduce

- Let's take a closer look at what we did!

```
# List of values  
values = [1, 2, 3, 4, 5, 6, 7]  
  
# Reducing function  
lambda x, y: 10*x+5
```

1234567

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's debugger!
- ~~collections~~ – specialized containers for data!
- ~~itertools~~ – better iterators!
- ~~functools~~ – functions on callable objects!
- `re` – regular expressions, pattern matching!
- `pickle` – object serialization!
- `pathlib` – object-oriented filesystem!
- `threading` – thread-based parallelism!

The `re` Package

Regular expressions and pattern matching!

Application Filtering for CS41 Using `re`

- In CS41, we got some truly wonderful applications (like all of yours!), and we got some that were... a little more interesting.

Article

[Talk](#)

Read

[Edit](#)

[View history](#)

Python (programming language)

From Wikipedia, the free encyclopedia

For other uses, see [Python](#).

Python is an [interpreted](#), [high-level](#), [general-purpose programming language](#). Created by [Guido van Rossum](#) and first released in 1991, Python's design philosophy emphasizes [code readability](#) with its notable use of [significant whitespace](#). Its language constructs and [object-oriented](#) approach aim to help programmers write clear, logical code for small and large-scale projects.^[27]

Application Filtering for CS41 Using `re`

- In CS41, we got some truly wonderful applications (like all of yours!), and we got some that were... a little more interesting.
- E.g.
 - **What do you study at Stanford?**
Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects
- Digging deeper, many of these types of responses came from non-Stanford emails. Can we use `re` to filter applications?

What are Regular Expressions?

- Regular expressions are patterns to which we can match strings.
- Special syntax (there's much more than this – this is just a primer!):
 - `.` - matches any character.
 - `*` - matches zero or more of the character preceding it. `a*` would match `""`, `"a"`, `"aaaaaaaa"`, etc.
 - `[]` – matches one of the characters between the brackets. So `a[bc]d` would match `"abd"` and `"acd"`.

Some Helpful `re` Commands

```
re.match(pattern, string)
```

*# Return MatchObject (which allows for
match processing) if string matches
pattern; None otherwise.*

```
re.search(pattern, string)
```

*# Scan through string looking for first
location where string matches
pattern. Use MatchObject.start to extract
the index.*

```
re.finditer(pattern, string)
```

*# Return an iterator yielding
MatchObjects of all non-overlapping
pattern matches in the string.*

```
re.sub(pattern, sub, string)
```

*# Return string created by substituting
sub parameter for all pattern matches
in string.*

Application Filtering for CS41 Using `re`

- Want to immediately filter out any application that does not:
 - Come from a valid email address.
 - Come from a *Stanford* email address.
- `re` does this work for you! Create a pattern, then check examples using `re.match`!

```
# Problem - application filtering for CS 41!

for email in applicant_emails:
    if re.match("[a-zA-Z0-9_]+@stanford.edu", email):
        # It's a valid application!
    else:
        # Filter it out of the process
```

Your Turn!

```
"""
```

Write a regular expression to match a phone number like
951 262-3062

Hint: `\d` captures `[0-9]` (i.e. any digit).

Hint: `\d{3}` captures three consecutive digits.

Hint: `[]` matches one of the characters between the brackets. So
`[123]` matches 1, 2, or 3.

```
"""
```

```
re.match(your_pattern, "951 262-3062")           #=> True  
re.match(your_pattern, "951.262.3062")           #=> False
```

```
# Done? Look through the re documentation and figure out how to  
# return the area code!
```

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's debugger!
- ~~collections~~ – specialized containers for data!
- ~~itertools~~ – better iterators!
- ~~functools~~ – functions on callable objects!
- ~~re~~ – regular expressions, pattern matching!
- `pickle` – object serialization!
- `pathlib` – object-oriented filesystem!
- `threading` – thread-based parallelism!

Attendance Form

Link: <http://iamhere.stanfordpython.com>

Code: STANDARD UNICORN

If you're not here, fill out this alternative form:

<http://bit.ly/2SGqOb8>



Announcements

- Assignment 2 (Quest for the Unicorn!) **due Friday, 11:59PM.**
 - Take photos of yourselves on your quest!
- Final project proposals are out!
 - Start thinking about what you want to work on. Remember – we're here to help! So approach us (or your TA) with any questions you may have
 - Proposals will be due **next Wednesday, 11:59PM.** More info on the course website!

The pickle Package

Object serialization!

What is pickling?

- Serializing a Python object into a bytes structure. Many synonymous names:
 - Serialization
 - Marshalling
 - Flattening
- In Python, this process is called **pickling**. The inverse process – rehydrating a bytes-object into a Python object in the hierarchy – is called **unpickling**.
- I've used this in machine learning – once I train a neural net, I pickle it so that when I need to run it on a new example, I can rehydrate the netural net without needing to retrain!

Warning!

- Only unpickle objects you trust!
- It is possible to construct pickle objects which execute arbitrary (read: potentially malicious!) code during unpickling.
- If you want to ensure the security of a pickled object, you can use [hmac](#), which performs cryptographic signing.
 - Come chat with us if you'd like to learn more!



Pickling a Python Object

- Most commonly, we write the pickled bytestring to a file. Similar syntax to file writing, except we use the `pickle.dump` function:

```
data = np.array([.....]) # Some object - this can be any Python object!  
  
with open('data_file_name.pickle', 'wb') as f:  
    pickle.dump(data, f)
```

- There are ways to do this in one line of code – but the context manager is the safe way! 😊
- To pickle to a string instead of a file, use the syntax `pickle.dumps(data_string)`.

Unpickling a Python Object

- Almost the same syntax as pickling, only we use the `pickle.load` function:

```
with open('data_file_name.pickle', 'rb') as f:  
    data = pickle.load(f)
```

- To pickle from a string instead of a file, use the syntax `pickle.loads(data_bytestring)`.

Today's Tour of the Standard Libraries

- ~~pdb~~ – Python's debugger!
- ~~collections~~ – specialized containers for data!
- ~~itertools~~ – better iterators!
- ~~functools~~ – functions on callable objects!
- ~~re~~ – regular expressions, pattern matching!
- ~~pickle~~ – object serialization!
- `pathlib` – object-oriented filesystem!
- `threading` – thread-based parallelism!

The `pathlib` Package

Object-oriented filesystem!

A Quick Comparison...

- Filesystem – "input.txt" stored in "in" folder; "output.txt" stored in "out" folder.
- Which of these is easier to parse?

```
import os

in_file = os.path.join(os.path.join(os.getcwd(), "in"), "input.txt")
out_file = os.path.join(os.path.join(os.getcwd(), "out"), "output.txt")
```

```
from pathlib import Path

in_file_1 = Path.cwd() / "in" / "input.txt"
out_file_1 = Path.cwd() / "out" / "output.txt"
```

The Path Object

- An object to represent the path to a file or directory.
- Constructor takes in directory path, generates attributes from scanning the directory.

```
from pathlib import Path

p = Path("path/to/file.txt")      # This can be any filepath!

p.is_dir()                        #=> False
p.is_file()                       #=> True
p.parts                           #=> ('/', 'path', 'to', 'file.txt')
p.parent()                        #=> PosixPath('/path/to')
p.Parent / "other_dir"           #=> PosixPath('/path/to/other_dir')
```

- Pass path objects into `open` just as you would a path string!

Today's Tour of the Standard Libraries

- ~~pdb~~ — Python's debugger!
- ~~collections~~ — specialized containers for data!
- ~~itertools~~ — better iterators!
- ~~functools~~ — functions on callable objects!
- ~~re~~ — regular expressions, pattern matching!
- ~~pickle~~ — object serialization!
- ~~pathlib~~ — object-oriented filesystem!
- ~~threading~~ — thread-based parallelism!

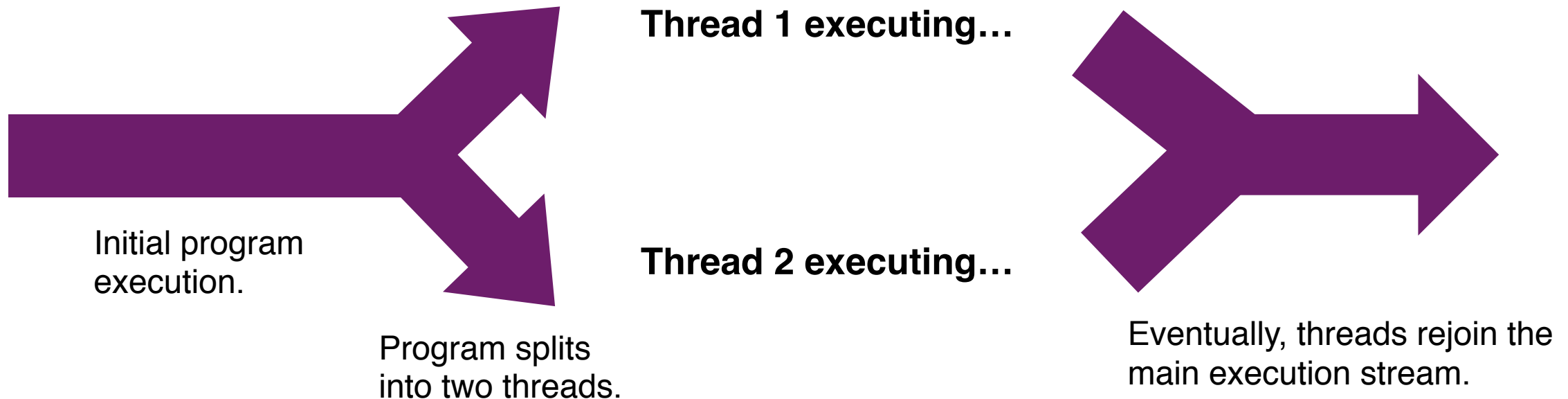
The `threading` Package

Concurrent execution!

What is a Thread?

What is Multithreading?

- A thread refers to a "thread of execution". Programs can split themselves into multiple threads.
 - Multiple threads within one program can run concurrently! This is called multithreading.



Why is Multithreading Important?

- Reducing "busywaiting".
- Example:
 - You're downloading 10 images using `requests`, each from a different server.
 - Data transfer is intermittent – milliseconds in which no data is received, followed by milliseconds in which data is received.
 - Each server's data transfer is independent of that of the others.
 - Without parallelism: download each image one at a time. Only once one image is received do you begin downloading the next.
 - With parallelism: whenever a server has data to download, take it; when it doesn't, download data from another server that does.
 - If each download is in its own thread, ~10x speedup!

Why is Multithreading Important?

Receiving Data, One Thread



Receiving Data, Multiple Threads



Focus on the processes that are currently being productive; when one is paused, focus on another.

A First Threading Example

```
"""
```

When we run this code, notice that the 5 seconds of sleeping time per thread is concurrent.

```
"""
```

```
import time
```

```
from threading import Thread
```

```
def sleep_thread(i):
```

```
    print("Thread {} is about to take a nap!".format(i))
```

```
    time.sleep(5)
```

```
    print("Thread {} has woken up from naptime!".format(i))
```

```
for i in range(10):
```

```
    t = Thread(target=sleep_thread, args=(i,))
```

```
    t.start()
```


Python's Deception

- In Python's implementation of `threading`, process do not actually run concurrently – they merely appear to. (Even if your machine has multiple cores!)
 - If you want true concurrency, Python has a `multiprocessing` library that is worth checking out!
- Despite this, `threading` is still useful!
 - Programs that wait for external events are good candidates for the use of threading. E.g. server applications, downloading, etc.
 - The operating system scheduler is good at running only those threads which are able to do work at the time.

A Second Threading Example

```
"""
Multithreaded downloading images from the web!
"""
from threading import Thread
from io import BytesIO
from PIL import Image
import requests

def download_image(url):
    img_bytes = BytesIO(requests.get(url).content)
    image = Image.open(img_bytes)
    image.show()

urls = [...] # List of urls from which to download
for url in urls:
    t = Thread(target=download_image, args=(url,))
    t.start()
```

Thread-Safety

- Multiple threads can access the same resource – this may be problematic if the resource is mutable.
- Scenario example:
 - Thread 1 adds data to a (global) list.
 - Thread 1 is taken off the CPU and Thread 2 is running.
 - Thread 2 accesses the list, overwrites Thread 1's data.
 - Thread 1 tries to access its data to do something with it, and it's the wrong data.
 - Catastrophe!!

Thread-Safety

- Multithreaded programs deal with these cases through the use of locks.
 - A lock can be acquired by a thread.
 - It must be released by the thread that acquired it before another thread can acquire the lock.
 - This enables protecting sections of our code prone to multithreading errors.

```
# Initialize this where all threads can access it – here we do so globally
my_lock = threading.Lock()

# Then within a thread function...
my_lock.acquire()
# Sensitive code
my_lock.release()
```

Joining Threads

- Back in the main thread of program execution, how do we know all our threads have finished?
- We use the `t.join()` function (where `t` is a thread) – `t.join()` waits for `t` to finish, then proceeds with the normal flow of execution.
- Usually this is placed in a for loop after spawning the threads.

```
# Add each thread to the list when you spawn threads
thread_list = []

# Wait for all threads to have finished
for t in thread_list:
    t.join()
    print("All threads have finished!")
```

Common One-Liners

- Some of these are likely review – but here are some common builtin expressions!

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

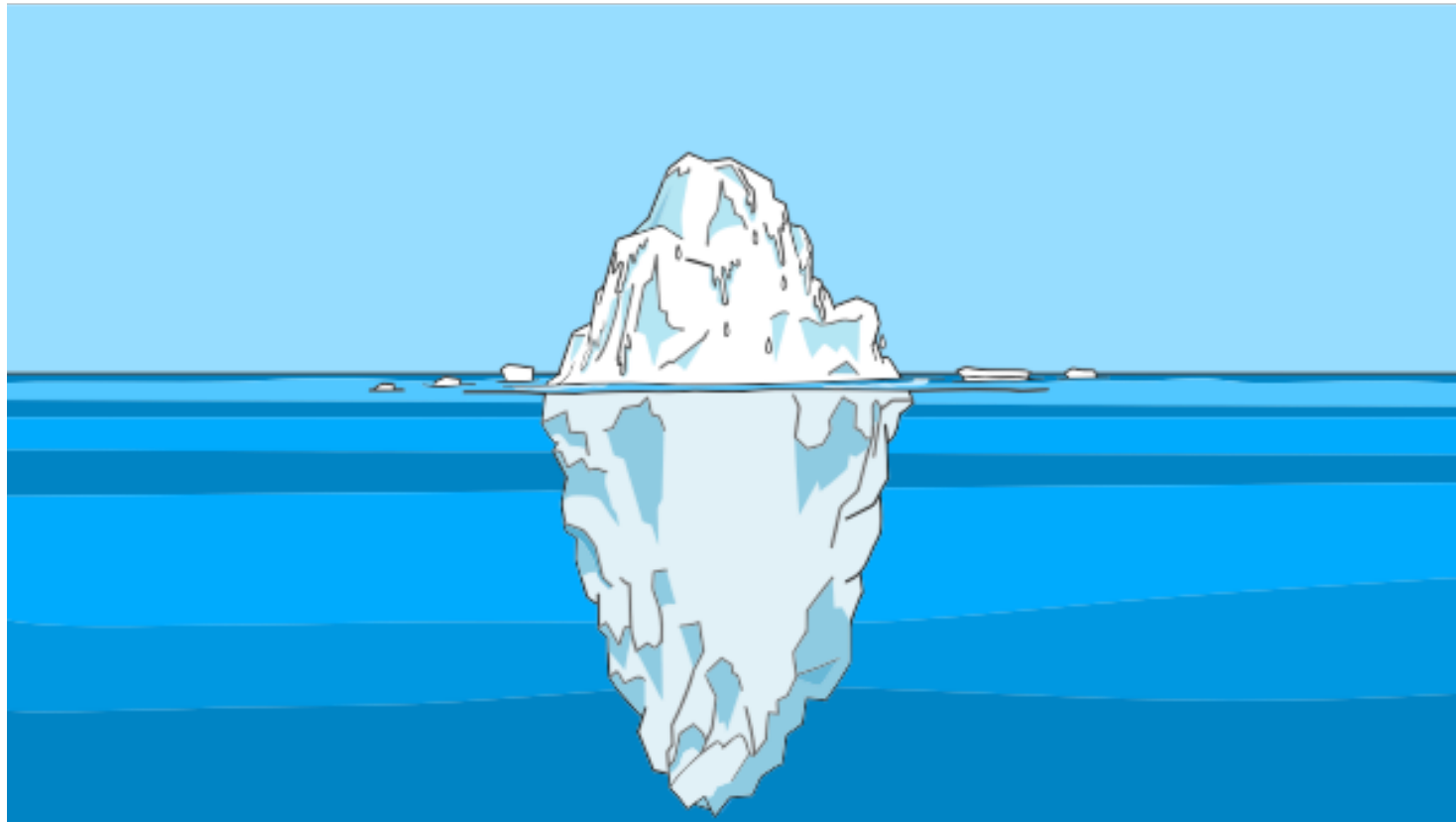
int('45')                   #=> 45
int(0x2a, 16)                #=> 42
int('1011', 2)               #=> 11
hex(42)                      #=> '0x2a'
bin(42)                      #=> '0b101010'

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

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

This is Only the Tip of the Iceberg...

- Python's Standard Library is **huge!**
 - Let's [take a peek!](#)



Summary

- Modules and Packages
- `pdb` – Python's Debugger!
- `collections` – Specialized Containers for Data!
- `itertools` – Better Iterators!
- `functools` – Functions on Callable Objects!
- `re` – Regular Expressions, Pattern Matching!
- `pickle` – Object Serialization!
- `pathlib` – Object-Oriented Filesystem!
- `threading` – Thread-Based Parallelism!

