Extra Topics in Python

None

Version 1.0, August 2025

Table of Contents

About Extra Topics in Python

Course Outline

Day 1

Chapter 1 Concurrency
Chapter 2 Type Hinting
Chapter 3 Introduction to Pandas
Chapter 4 Introduction to Matplotlib

The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **pyjpmc-max**. See the setup guides for details.

What's in the files?

pyjpmc-max contains all files necessary for the class
pyjpmc-max/EXAMPLES/ contains the examples from the course manuals.
pyjpmc-max/ANSWERS/ contains sample answers to the labs.
pyjpmc-max/DATA/ contains data used in examples and answers
pyjpmc-max/SETUP/ contains any needed setup scripts (may be empty)
pyjpmc-max/TEMP/ initially empty; used by some examples for output files

The following folders *may* be present:

pyjpmc-max/BIG_DATA/ contains large data files used in examples and answers
pyjpmc-max/NOTEBOOKS/ Jupyter notebooks for use in class
pyjpmc-max/LOGS/ initially empty; used by some examples to write log files

The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```
import sys   # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1]  # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

```
['/Users/jstrick/curr/courses/python/common/examples/cmd_line_args.py', 'apple', 'mango',
'123']
name is apple
```


Appendices

Classroom etiquette

Remote learning

- Mic off when you’re not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn’t know you need help unless you let them know via voice or chat.
- It’s ok to ask for help a lot.
    - Ask questions. Ask questions. Ask questions.
    - ***INTERACT*** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We’re all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don’t have to wait for a lab or break, but please try not to disturb others.

> Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

Chapter 1: Concurrency

Objectives

- Understand concurrency concepts
- Differentiate between threads, processes, and async
- Know when threads benefit your program
- Learn the limitations of the GIL
- Create a threaded application
- Use the multiprocessing module
- Develop a multiprocessing application

Concurrency

- Running more than one function concurrently
- Three main ways to achieve it
  - threading
  - multiple processes
  - asynchronous communication
- All supported in standard library

Computer programs spend a lot of their time doing nothing. This occurs when the CPU is waiting for the relatively slow disk subsystem, network stack, or other hardware to fetch data.

Some applications can achieve more throughput by taking advantage of this slack time by seemingly doing more than one thing at a time. With a single-core computer, this doesn’t really happen; with a multicore computer, an application really can be executing different instructions at the same time. This is called multiprogramming or *concurrency*.

The three main ways to implement multiprogramming are threading, multiprocessing, and asynchronous communication:

Threading subdivides a single process into multiple subprocesses, or threads, each of which can be performing a different task. Threading in Python is good for IO-bound applications, but does not increase the efficiency of compute-bound applications.

Multiprocessing forks (spawns) new processes to do multiple tasks. Multiprocessing is good for both CPU-bound and IO-bound applications.

Asynchronous communication uses an event loop to poll multiple I/O channels rather than waiting for one to finish. Asynch communication is good for IO-bound applications, and can be faster than the other approaches.

The standard library supports all three.

Threads

- Like processes (but lighter weight)
- Use fewer resources (memory and CPU)
- Process can create one or more additional threads
- Similar to creating new processes with fork()

Modern operating systems (OSs) use time-sharing to manage multiple programs which appear to the user to be running simultaneously. Assuming a standard machine with only one CPU, that simultaneity is only an illusion, since only one program can run at a time, but it is a very useful illusion. Each program that is running counts as a process. The OS maintains a process table, listing all current processes. Each process will be shown as currently being in either Run state or Sleep state.

A thread is like a process. A thread might even be a process, depending on the implementation. In fact, threads are sometimes called “lightweight” processes, because threads occupy much less memory, and take less time to create, than do processes.

A process can create any number of threads. This is similar to a process calling the fork() function. The process itself is a thread, and could be considered the "main" thread.

Just as processes can be interrupted at any time, so can threads.

The Python Thread Manager

- Python uses underlying OS's threads
- Alas, the GIL – Global Interpreter Lock
- Only one thread runs at a time
- Python interpreter controls end of thread's turn
- Cannot take advantage of multiple processors

Python "piggybacks" on top of the OS's underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the OS's thread list.

However, Python imposes further structure on top of the OS threads. Most importantly, there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that (a) only one thread runs at a time, and (b) that the ending of a thread's turn is controlled by the Python interpreter rather than the external event of the hardware timer interrupt.

The fact that the GIL allows only one thread to execute Python bytecode at a time simplifies the Python implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. The takeaway is that Python does not currently take advantage of multi-processor hardware.

GIL is pronounced "jill", according to Guido__

For a thorough discussion of the GIL and its implications, see http://www.dabeaz.com/python/UnderstandingGIL.pdf.

The `threading` Module

- Provides basic threading services
- Also provides locks, events, and other tools
- Three ways to use threads
  - Instantiate `Thread` with a function
  - Subclass `Thread`
  - Use thread pool from `multiprocessing`

The `threading` module provides basic threading services for Python programs. The usual approach is to subclass `threading.Thread` and define a `run()` method that does the thread's work.

Threads for the impatient

- No class needed (created "behind the scenes")
- For simple applications

For many threading tasks, all you need is a run() method and maybe some arguments to pass to it.

For simple tasks, you can just create an instance of Thread.

Constructor arguments

`target`

Use the `target` argument to pass in the function for the thread to run.

`args`

Use the `args` argument to pass in a tuple of arguments for the target function.

`name`

Use to set the name of the thread. This is an arbitrary string

`kwargs`

Use `kwargs` to provide a dictionary of named arguments to the target function.

Always pass arguments to `Thread()` by name.

Example

thr_noclass.py

```
from threading import Thread, Lock
import random
import time

STDOUT_LOCK = Lock()

def my_task(num):  # function to run in each thread
    time.sleep(random.randint(1, 3))
    with STDOUT_LOCK:
        print(f"Hello from thread {num}")

for i in range(16):
    t = Thread(target=my_task, args=(i,))  # create thread
    t.start()  # launch thread

print("Done.")  # "Done" is printed immediately -- the threads are "in the background"
```

thr_noclass.py

```
Done.
Hello from thread 2
Hello from thread 3
Hello from thread 11
Hello from thread 15
Hello from thread 1
Hello from thread 0
Hello from thread 13
Hello from thread 5
Hello from thread 14
Hello from thread 8
Hello from thread 4
Hello from thread 6
Hello from thread 7
Hello from thread 9
Hello from thread 10
Hello from thread 12
```

Subclassing `Thread`

- Must call base class constructor
- Must define run()
- Can implement helper methods

A thread class is a class that starts a thread, and performs some task. Such a class can be repeatedly instantiated, with different parameters, and then started as needed.

The class can be as elaborate as the business logic requires. There are only two rules: the class must call the base class’s constructor, and it must define a `run()` method. Other than that, the `run()` method can do pretty much anything it wants to.

The best way to invoke the base class constructor is to use `super().init()`.

The `run()` method is invoked when you call the `start()` method on the thread object. The `start()` method does not take any parameters; `run()` has no parameters as well.

Any per-thread arguments can be passed into the constructor when the thread object is created.

Example

thr_simple.py

```
from threading import Thread, Lock
import random
import time

STDOUT_LOCK = Lock()

class SimpleThread(Thread):
    def __init__(self, num):
        super().__init__()  # call base class constructor -- REQUIRED
        self._threadnum = num

    def run(self):  # the function that does the work in the thread
        time.sleep(random.randint(1, 3))
        with STDOUT_LOCK:
            print(f"Hello from thread {self._threadnum}")


for i in range(16):
    t = SimpleThread(i)  # create the thread
    t.start()  # launch the thread

print("Done.")
```

thr_simple.py

```
Done.
Hello from thread 0
Hello from thread 1
Hello from thread 3
Hello from thread 13
Hello from thread 8
Hello from thread 5
Hello from thread 4
Hello from thread 6
Hello from thread 7
Hello from thread 10
Hello from thread 14
Hello from thread 2
Hello from thread 9
Hello from thread 11
Hello from thread 12
```

```
Hello from thread 15
```


Variable sharing

- Variables declared before thread starts are shared
- Variables in the thread function are local

A major difference between ordinary processes and threads how variables are shared.

Each thread has can have its own local variables, just as is the case for any function. However, global variables that existed in the program before a thread was spawned are accessible by the thread.

Write access to global variables should be guarded by locks.

Example

thr_locking.py

```
import threading
import random
import time

WORD_LIST = 'apple banana mango peach papaya cherry lemon watermelon'.split()

MAX_SLEEP_TIME = 3
RESULT_LIST = []  # the threads will append words to this list
RESULT_LIST_LOCK = threading.Lock()  # generic locks
STDOUT_LOCK = threading.Lock()  # generic locks

class SimpleThread(threading.Thread):
    def __init__(self, word):  # thread constructor
        super().__init__()  # be sure to call parent constructor
        self._word = word   # value is passed into thread for processing

    def run(self):  # function invoked by each thread
        time.sleep(random.randint(1, MAX_SLEEP_TIME))

        with STDOUT_LOCK:  # acquire lock and release when finished
            print(f"Starting thread {self.ident} with value {self._word}")

        with RESULT_LIST_LOCK:  # acquire lock and release when finished
            RESULT_LIST.append(self._word.upper())

all_threads = []  # make list ("pool") of threads (but see Pool later in chapter)
for random_word in WORD_LIST:  # inefficiently creating one thread per word...
    t = SimpleThread(random_word)  # create thread
    all_threads.append(t)  # add thread to "pool"
    t.start()  # start thread

print("All threads launched...")

for t in all_threads:
    t.join()  # wait for thread to finish

print(RESULT_LIST)
```


thr_locking.py

```
All threads launched...
Starting thread 123145384697856 with value mango
Starting thread 123145351118848 with value apple
Starting thread 123145418276864 with value papaya
Starting thread 123145451855872 with value lemon
Starting thread 123145435066368 with value cherry
Starting thread 123145367908352 with value banana
Starting thread 123145401487360 with value peach
Starting thread 123145468645376 with value watermelon
['MANGO', 'APPLE', 'PAPAYA', 'LEMON', 'CHERRY', 'BANANA', 'PEACH', 'WATERMELON']
```

Thread coordination

- Can't assign to immutable variables
- Use `threading.Event()`
  - `event.set()`
  - `event.wait()`
  - `event.clear()`

Sometimes a thread will need to synchronize with or signal another thread. This can get a little messy when using shared variables.

The `threading` module provides an `Event` object to make this simpler.

An Event object can be either set or cleared. Some other thread can wait for the event to be set, or check to see whether it is set.

This can be useful for setting a timer, or for telling a thread to start processing data.

Example

thr_signal.py

```
from threading import Thread, Event
import time

STOP_TASK = Event()

def do_something():
    for i in range(50):
        if STOP_TASK.is_set():
            break
        print(f'{i}-', end='', flush=True)
        time.sleep(.5)

def interrupt():
    time.sleep(10)
    print("STOPPING!")
    STOP_TASK.set()

if __name__ == "__main__":
    t = Thread(target=interrupt)
    t.start()  # start thread, which will set the event 10 seconds later
    do_something()  # start function, which will detect the event in about 10 seconds
```

thr_signal.py

```
0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-STOPPING!
```

Example

thr_sync.py

```
from functools import partial
from threading import Thread, Event, Lock
import time

a_ready = Event()
stdout_lock = Lock()

def pr(*args):
    with stdout_lock:
        print(*args, end='', flush=True)

def divisible_by_n(value, n):
    if value == 0:
        return True
    if (value > 0) and ((value % n) == 0):
        return True
    return False

class ThreadA(Thread):

    def run(self):
        for i in range(1, 50):
            pr(f"A{i}")
            if divisible_by_n(i, 10):
                a_ready.set()  # notify b
                pr("/setting/")
            elif divisible_by_n(i, 5):
                a_ready.clear()  # stop notifying b
                pr("/clearing/")
            time.sleep(.1)
        pr("/setting/")
        a_ready.set()  # notify b and let b finish


class ThreadB(Thread):

    def run(self):
        for i in range(1, 50):
            a_ready.wait()  # wait until event is set by ThreadA
            pr(f"B{i}")
            time.sleep(.1)
```



```
t_a = ThreadA()
t_a.start()
t_b = ThreadB()
t_b.start()
t_a.join()
t_b.join()
print()
```

thr_sync.py

```
A1A2A3A4A5/clearing/A6A7A8A9A10/setting/B1A11B2A12B3A13B4A14B5A15/clearing/A16A17A18A19A2
0/setting/B6B7A21A22B8A23B9A24B10A25B11/clearing/A26A27A28A29A30/setting/B12A31B13A32B14A
33B15A34B16B17A35/clearing/A36A37A38A39A40/setting/B18B19A41B20A42B21A43B22A44B23A45/clea
ring/A46A47A48A49/setting/B24B25B26B27B28B29B30B31B32B33B34B35B36B37B38B39B40B41B42B43B44
B45B46B47B48B49
```

Using queues

- Queue contains a list of objects
- Sequence is FIFO
- Worker threads can pull items from the queue
- `queue.Queue` structure has builtin locks

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

The queue must be guarded with locks. Python provides the Queue module to take care of all the lock creation, locking and unlocking, and so on, so that you don’t have to bother with it.

The `queue` module provides a thread-safe `Queue` object that does not need to be guarded with locks. It is a FIFO sequence of values. You can `.put()` values in one end and `.get()` them from the other.

Example

thr_queue.py

```
import random
import queue
from threading import Thread, Lock as tlock
import time

NUM_ITEMS = 30000
POOL_SIZE = 128

word_queue = queue.Queue(0)  # initialize empty queue

shared_list = []
shared_list_lock = tlock()  # create locks
stdout_lock = tlock()  # create locks


class RandomWord():  # define callable class to generate words
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in.readlines()]
        self._num_words = len(self._words)

    def __call__(self):
        return self._words[random.randrange(0, self._num_words)]
```

```
class Worker(Thread):  # worker thread

    def __init__(self):  # thread constructor
        Thread.__init__(self)

    def run(self):  # function invoked by thread
        while True:
            try:
                s1 = word_queue.get(block=False)  # get next item from thread
                s2 = s1.upper() + '-' + s1.upper()
                with shared_list_lock:  # acquire lock, then release when done
                    shared_list.append(s2)

            except queue.Empty:  # when queue is empty, it raises Empty exception
                break


# fill the queue
random_word = RandomWord()
for i in range(NUM_ITEMS):
    w = random_word()
    word_queue.put(w)

start_time = time.ctime()

# populate the threadpool
pool = []
for i in range(POOL_SIZE):
    w = Worker()  # add thread to pool
    w.start()  # launch the thread
    pool.append(w)

for t in pool:
    t.join()  # wait for thread to finish

end_time = time.ctime()

print(shared_list[:20])

print(start_time)
print(end_time)
```

thr_queue.py

```
['LEADLESS-LEADLESS', 'OVERBURNT-OVERBURNT', 'CARRIES-CARRIES', 'PICNICKERS-PICNICKERS',
'LANDSCAPING-LANDSCAPING', 'SHEARLING-SHEARLING', 'ESTIMATIVE-ESTIMATIVE', 'HOODOOISM-
HOODOOISM', 'OOLOGIST-OOLOGIST', 'PANICUM-PANICUM', 'SHIVERING-SHIVERING', 'REENDOWS-
REENDOWS', 'LIKERS-LIKERS', 'SAGGAR-SAGGAR', 'VERSTE-VERSTE', 'TRIENES-TRIENES',
'BACCHII-BACCHII', 'ARITHMETICAL-ARITHMETICAL', 'DETERRABLE-DETERRABLE', 'SAHIWALS-
SAHIWALS']
Thu Aug 14 08:03:18 2025
Thu Aug 14 08:03:18 2025
```


Debugging threaded Programs

- Trickier than non-threaded programs
- Context changes abruptly
- Use `breakpoint()`

Debugging threads can be tricky. If you're used to debugging normal programs, it can be surprising to suddenly jump into the code of a different thread. Debugging deadlocks can be very hard.

Another problem which sometimes occurs is that if you issue a **next** command in your debugging tool, you may end up inside the internal threads code. In such cases, use a **continue** command to get back to your code.

The basic PDB debugger may not work for debugging like this:

```
pdb.py mythreadedprogram.py
```

This is because threads will not inherit the `pdb` process from the main thread. While you can run the debugger on the main program, you won't be able to set breakpoints in threads.

To get around this issue, use `breakpoint()` in your actual code; this builtin function will invoke PDB when it is called. It is a way of setting breakpoints in your code.

When using `breakpoint()`, execute your program normally, not with `pdb`, and it will stop at the first instance of `breakpoint()` and invoke `pdb`.

Prior to version 3.7, import `pdb` and use `pdb.set_trace()` to get the same behavior as `breakpoint()`

The multiprocessing module

- Drop-in replacement for the threading module
- Doesn't suffer from GIL issues
- Provides interprocess communication
- Provides process (and thread) pooling

The `multiprocessing` module can be used as a replacement for` threading`. It uses processes rather than threads to spread out the work to be done. While the entire module doesn't use the same API as threading, `multiprocessing.Process` object is a drop-in replacement for `threading.Thread`. Both use `.run()` as the overridable method that does the work, and both use `.start()` to launch. The syntax is the same to create a process without using a class:

```
def myfunc(filename):
      pass

p = Process(target=myfunc, args=('/tmp/info.dat', ))
```

This solves the GIL issue, but the trade-off is that it's a little slower, and slightly more complicated for tasks (processes) to communicate. However, the module does the heavy lifting of creating pipes to share data.

The `Manager` class provided by multiprocessing allows you to create shared variables, as well as locks for them, which work across processes.

On Windows, processes must be started in the `if __name__ == __main__:` block, or they will not work.

Example

multi_processing.py

```
import sys
import random
from multiprocessing import Manager, Lock, Process, Queue, freeze_support
from queue import Empty
import time

NUM_ITEMS = 25000  # set some constants
POOL_SIZE = 64
```



```
class RandomWord():  # callable class to provide random words
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in]
        self._num_words = len(self._words)

    def __call__(self):  # will be called when you call an instance of the class
        return self._words[random.randrange(0, self._num_words)]


class Worker(Process):  # worker class -- inherits from Process

    def __init__(self, name, queue, lock, result):  # initialize worker process
        Process.__init__(self)
        self.queue = queue
        self.result = result
        self.lock = lock
        self.name = name

    def run(self):  # do some work -- will be called when process starts
        while True:
            try:
                word = self.queue.get(block=False)  # get data from the queue
                word = word.upper()  # modify data
                with self.lock:
                    self.result.append(word)  # add to shared result

            except Empty:  # quit when there is no more data in the queue
                break


if __name__ == '__main__':
    if sys.platform == 'win32':
        freeze_support()

    word_queue = Queue()  # create empty Queue object

    manager = Manager()  # create manager for shared data
    shared_result = manager.list()  # create list-like object to be shared across all
processes
    result_lock = Lock()  # create locks

    random_word = RandomWord()  # create callable RandomWord instance
    for i in range(NUM_ITEMS):
        w = random_word()
        word_queue.put(w)  # fill the queue
```

```
    start_time = time.ctime()

    pool = []  # create empty list to hold processes
    for i in range(POOL_SIZE):  # populate the process pool
        worker_name = f"Worker {i:03d}"
        w = Worker(worker_name, word_queue, result_lock, shared_result)  # create worker
process
        #
        w.start()  # actually start the process -- note: in Windows, should only call
X.start() from main(), and may not work inside an IDE
        pool.append(w)  # add process to pool

    for t in pool:
        t.join()  # wait for each queue to finish

    end_time = time.ctime()

    print((shared_result[-50:]))  # print last 50 entries in shared result
    print(len(shared_result))
    print(start_time)
    print(end_time)
```

multi_processing.py

```
['DETERMINACY', 'SUBJECTIVIZES', 'APING', 'PUNDITRIES', 'CRUX', 'HEADRACES',
'UNDISTINGUISHED', 'CADENT', 'ADAPTATION', 'SUPERFINE', 'DEMASTS', 'ARS', 'RAVISHED',
'ERYTHROBLASTS', 'HOLOGRAM', 'CRICKETS', 'THAE', 'AGLYCONE', 'DEIXISES', 'LANDHOLDER',
'CARNEY', 'SANGH', 'ATARAXIA', 'PATTERN', 'TOOTHIER', 'CONVENES', 'QUESADILLA',
'FIDGETS', 'AMOROSO', 'STEREOTYPIC', 'MUTELY', 'EVIDENCED', 'SHIKSES', 'DRIFTPIN',
'EXCLAIMED', 'BIBCOCKS', 'BALLADRY', 'MICROMERES', 'DENTITIONS', 'SYLLABIFIED', 'ROADS',
'CARTOONISTS', 'SUBDUCTS', 'RESPIRES', 'TRIPLICATION', 'AWNED', 'PAPERBOUNDS',
'CATECHOLAMINES', 'CYANOACRYLATE', 'CALIBRATORS']
25000
Thu Aug 14 08:03:18 2025
Thu Aug 14 08:03:23 2025
```


Using pools

- Provided by `multiprocessing` and `multiprocessing.dummy`
- Both thread and process pools
- Simplifies multiprogramming tasks

For many multiprocessing tasks, you want to process a list (or other iterable) of data and do something with the results. This is easily accomplished with the `Pool` class provided by the `multiprocessing` module.

This object creates a pool of *n* processes. Call the `.map()` method with a function that will do the work, and an iterable of data. `.map()` will return a list of results in the same order as the original data.

For a thread pool, import `Pool` from `multiprocessing.dummy`. It works exactly the same, but creates threads.

Example

proc_pool.py

```
import random
from multiprocessing import Pool

POOL_SIZE = 32  # number of processes

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in]  # read word file into a list, stripping off


random.shuffle(WORDS)  # randomize word list



def my_task(word):  # actual task
    return word.upper()



if __name__ == '__main__':
    ppool = Pool(POOL_SIZE)  # create pool of POOL_SIZE processes

    WORD_LIST = ppool.map(my_task, WORDS)  # pass wordlist to pool and get results; map
assigns values from input list to processes as needed

    print(WORD_LIST[:20])  # print last 20 words

    print(f"Processed {len(WORD_LIST)} words.")
```

proc_pool.py

```
['CHOLINES', 'HANTLES', 'OUTRIDING', 'VILIPENDING', 'TURBIDIMETER', 'MUMMIFICATION',
'TERMINUSES', 'CICOREE', 'SKIDDY', 'INVIGORATE', 'LAICISES', 'UNPAID', 'JEFE',
'MYELOPATHIC', 'OVERSWEETENED', 'ANTIQUARIANS', 'INCISIVELY', 'PRINTS', 'PREPASTES',
'RINGHALSES']
Processed 173462 words.
```

Example

thr_pool.py

```
from multiprocessing.dummy import Pool # get the thread pool object

POOL_SIZE = 32 # set # of threads to create

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in] # get list of 175K words

def my_task(word):  # function to apply to each element
    return word.upper()

thread_pool = Pool(POOL_SIZE) # create pool

word_list = thread_pool.map(my_task, WORDS) # map elements across all threads

print(word_list[:20])

print(f"Processed {len(word_list)} words.")
```

thr_pool.py

```
['AA', 'AAH', 'AAHED', 'AAHING', 'AAHS', 'AAL', 'AALII', 'AALIIS', 'AALS', 'AARDVARK',
'AARDVARKS', 'AARDWOLF', 'AARDWOLVES', 'AARGH', 'AARRGH', 'AARRGHH', 'AAS', 'AASVOGEL',
'AASVOGELS', 'AB']
Processed 173462 words.
```

Example

thr_pool_mw.py

```
from multiprocessing.dummy import Pool  # .dummy has Pool for threads
import requests
import time

POOL_SIZE = 8

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  # base url
of site to access

with open('dictionaryapikey.txt') as api_key_in:
    API_KEY = api_key_in.read().rstrip()  # get credentials

SEARCH_TERMS = [  # terms to search for; each thread will search some of these terms
    'wombat', 'pine marten', 'python', 'pearl',
    'sea', 'formula', 'translation', 'common',
    'business', 'frog', 'muntin', 'automobile',
    'green', 'connect','vial', 'battery', 'computer',
    'sing', 'park', 'ladle', 'ram', 'dog', 'scalpel',
    'emulsion', 'noodle', 'combo', 'battery'
]
def main():
    total_times = {}
    for function in get_data_threaded, get_data_serial:
        start_time = time.perf_counter()
        results = function()
        for search_term, result in zip(SEARCH_TERMS, results):  # iterate over results,
mapping them to search terms
            print(search_term.upper(), end=": ")
            if result:
                print(result)
            else:
                print("** no results **")
        total_times[function.__name__] = time.perf_counter() - start_time
        print('-' * 60)

    for function_name, elapsed_time in total_times.items():
        print(f"{function_name} took {elapsed_time:.2f} seconds")



def fetch_data(term):  # function invoked by each thread for each item in list passed to
map()
    try:
        response = requests.get(
```



```
            BASE_URL + term,
            params={'key': API_KEY},
        )  # make the request to the site
    except requests.HTTPError as err:
        print(err)
        return []
    else:
        data = response.json()  # convert JSON to Python structure
        parts_of_speech = []
        for entry in data: # loop over entries matching search terms
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = entry.get("fl")
                    if part_of_speech:
                        parts_of_speech.append(part_of_speech)
        return sorted(set(parts_of_speech))  # return list of parsed entries matching
search term


def get_data_threaded():
    p = Pool(POOL_SIZE)  # create pool of POOL_SIZE threads
    return p.map(fetch_data, SEARCH_TERMS)  # launch threads, collect results

def get_data_serial():
    return [fetch_data(w) for w in SEARCH_TERMS]


if __name__ == '__main__':
    main()
```

…

Alternatives to `POOL.map()`

- map elements of iterable to multiple task function arguments
- map elements asynchronously
- apply task function to a single value
- apply task function to a single value asynchronously

There are some alternative methods to `Pool.map()`. These apply functions to the data in different patterns, and can be run asynchrously as well.

Table 1. Pool methods

method	multiple args	concurrent	blocks until done	results ordered
`map()`	no	yes	yes	yes
`apply()`	yes	no	yes	no
`map_async()`	no	yes	no	yes
`apply_async()`	yes	yes	no	no

.

Alternatives to threading and multiprocessing

- `asyncio`
- `Twisted`

Threading and forking are not the only ways to have your program do more than one thing at a time. Another approach is asynchronous programming. This technique putting events (typically I/O events) in a list, or queue, and starting an event loop that processes the events one at a time. If the granularity of the event loop is small, this can be as efficient as multiprogramming.

Async

Asynchronous programming is only useful for improving I/O throughput, such as networking clients and servers, or scouring a file system. Like threading (in Python), it will not help with raw computation speed.

The `asyncio` module in the standard library provides the means to write asynchronous clients and servers.

Twisted

The **Twisted** framework is a large and well-supported third-party module that provides support for many kinds of asynchronous communication. It has prebuilt objects for servers, clients, and protocols, as well as tools for authentication, translation, and many others. Find Twisted at twistedmaxtrix.com/trac.

See the files named `consume_omdb*.py` and `omdblib.py` in EXAMPLES for examples comparing single-threaded, multi-threaded, multi-processing, and async versions of the same program. There are also examples using `concurrent.futures`, an alternate interface for creating thread or process pools.

Chapter 1 Exercises

For each exercise, ask the questions: Should this be multi-threaded or multi-processed? Distributed or local?

Exercise 1-1 (apod.py, apod_downloads.py)

Background

NASA provides many APIs for downloading astronomical images and information. One of these is the APOD (Astronomy Picture Of the Day).

The `apod` module in the root folder provides a function named `fetch_apod()` to fetch one APOD by date. The format for the date is `YYYY-MM-DD'` It downloads the image and saves it to a local file. The function returns `True` for a successful download, `False` otherwise. Some dates will return `False` if the APOD is not an image (it might be a **Youtube** link) or if the request times out. You can ignore those issues.

The `apod` module uses a demo-only API key which has usage restrictions. If you just want to run the script to see what it does, the demo key is sufficient. For writing your own script, go to https://api.nasa.gov/index.html#signUp to get a personal API key. The only personal information it requires is an email address. Replace "DEMO_KEY" with your personal API key in the module.

Exercise

Start with the existing script `apod_downloads.py` in the root folder. This script uses the `apod` module to download the NASA APOD for the each day of each January 2023. Note how long it takes as written.

Update the script to be concurrent using a thread pool. Put all the code in the `main()` function.

Compare the speed of the original to the concurrent version.

Exercise 1-2 (folder_scanner.py)

Write a program that takes in a directory name on the command line, then traverses all the files in that directory tree and prints out a count of:

- how many total files
- how many total lines (count '\n')
- how many bytes (len() of file contents)

HINT: Use either a thread or a process pool in combination with **os.walk()**.

Chapter 2: Type Hinting

Objectives

- Learn how to annotate variables and parameters
- Find out what the type hints do **not** provide
- Employ the `typing` module to annotate collections
- Use `mypy` for type checking
- Correctly annotate multiple or special types

Type Hinting

Python supports optional type hinting of variables, function parameters, and return values. While the interpreter ignores these hints, they are useful in several ways:

- They make your code more self-documenting
- External static code analyzers can tell you about type mismatches, which can avoid bugs
- Documentation tools can extract types

Variables

Types may be specified with the declaration of a variable. It is not necessary to assign a value.

```
count: int = 0
file_path: str
values: list[float]
```

> declaring a variable with a type hint does not create the variable. If you try to use the variable before assigning a value, it will raise an error.

```
# Valid Python, type hint mismatch (ignored when program is run)
valid: dict = (3, 'hello')
```


Example

hints_variables.py

```
a: str
a = "abc"
a = 123
a = 123.456
print(f"{a = }")

b: float
b = "abc"
b = 123
b = 123.456
print(f"{b = }")
```

hints_variables.py

```
a = 123.456
b = 123.456
```

Functions

Python functions use a `->` to indicate a return type; function parameters are annotated with type information in the same way as variables.

Argument lists (optional arguments) and named argument lists may be type-hinted, the values are all expected to be of that type. For keyword arguments, the keys are still strings; only the values get the type hint.

Example

hints_functions.py

```
def shout(word: str, times: int = 1) -> str:
    return word.upper() * times

a: str = shout('Python')
print(f"{a = }")
b: list[float] = shout('Python', 3)
print(f"{b = }")
print()

def read_files(*file_paths: str):
    for file_path in file_paths:
        print(f"Opening {file_path}")
        with open(file_path) as file_in:
            pass

read_files('../DATA/mary.txt', '../DATA/parrot.txt')
print()

def shout_various(**kwargs: int) -> None:
    for word in kwargs:
        print(word.upper() * kwargs[word])

shout_various(python=10, perl=1, c=3)
```

hints_functions.py

```
a = 'PYTHON'
b = 'PYTHONPYTHONPYTHON'

Opening ../DATA/mary.txt
Opening ../DATA/parrot.txt

PYTHONPYTHONPYTHONPYTHONPYTHONPYTHONPYTHONPYTHONPYTHONPYTHON
PERL
CCC
```

Remember that type hinting is optional, and not enforced by the Python interpreter.

Static Type Checking

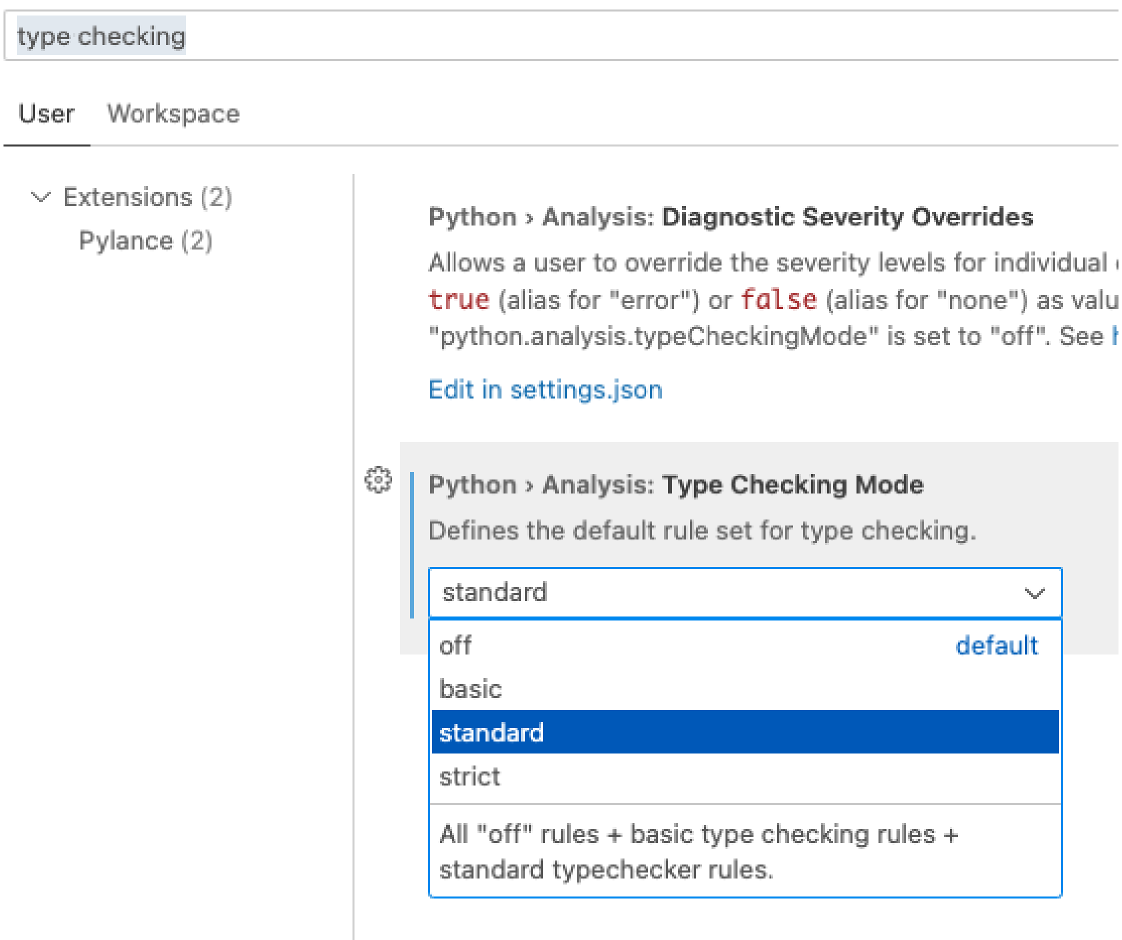
If these type hints are not used by the Python interpreter, how are they useful? While the Python interpreter does not (currently) use the type hints in any way, static analysis tools do. They check code *before* it is executed.

IDEs

Some IDEs will perform type checking as you write code.

Visual Studio Code

Type checking is off by default. To turn it on, go to **File › Preferences › Settings**. Search for "type checking". You can set type checking mode to "off", "basic", "standard", or "strict". For most programmers, "basic" or "standard" is a good choice.



TIP: On Macs, start with **Code › Settings › Settings** menu rather than the **File** menu.

PyCharm

Type checking is always on in PyCharm.

Spyder

Spyder does not currently support type checking.

MyPy

The most common tool for static type analysis, other than IDEs, is the `mypy` module. This is a "third-party" module (not part of the standard distribution) and can be installed with `pip`.

```
> pip install mypy
> python -m mypy hints_sample.py
```

The `mypy` module will scan the code (technically, AST of the code) and try to determine, at "compile" time, whether the types expected and used match up correctly. It will emit errors when it detects static typing problems in the code.

`mypy` even supports scanning the inline arbitrary code that may be present in a format-string literal.

```
word: str = 'hello'
# mypy will report an error on the next line
print(f'{word + 3})
```

`mypy` will emit errors, warnings, and notes of what it finds. While the output is quite configurable, most projects would benefit from fixing any and all issues found by `mypy`.

Example

python -m mypy hints_variables.py

```
hints_variables.py:4: error: Incompatible types in assignment (expression has type "int",
variable has type "str")  [assignment]
hints_variables.py:5: error: Incompatible types in assignment (expression has type
"float", variable has type "str")  [assignment]
hints_variables.py:9: error: Incompatible types in assignment (expression has type "str",
variable has type "float")  [assignment]
Found 3 errors in 1 file (checked 1 source file)
```

python -m mypy hints_functions.py

```
hints_functions.py:6: error: Incompatible types in assignment (expression has type "str",
variable has type "list[float]")  [assignment]
Found 1 error in 1 file (checked 1 source file)
```

Hints for collections

Lists

For lists, specify the type of all members of the list.

```
# Expects a list of strings
def process(record: list[str]) -> None:
    ...
```

```
# Expects a list of integers
def process(record: list[int]) -> None:
    ...
```

Tuples

`Tuple` objects generally specify exactly which type each positional value is, such as `Tuple[str, int, str]`. Tuples of arbitrary length (but the same type throughout) may be specified using the `Ellipsis` object.

```
def ziptuple(words: Tuple[str, ...], times: Tuple[int, ...]) -> Generator[str, None,
None]:
    for s, i in zip(words, times):
        yield s * i
```

```
# Expects a three-tuple with types of str, int, and float
def process(record: tuple[str, int, float]) -> None:
    ...
```


Dictionaries

With dictionaries, you can specify the types of both keys and value.

```
# a list of dictionaries
airports_by_state: dict[str, list]
```

Sets

Specify the value of all elements of the set.

```
# a set of floats
values = set[float]
```

Hints for unions (multiple types) and optional parameters

Some annotations need to include more than one type. This is handled by *unions*. Some annotations are for optional parameters, this is handled by *optional* types.

Union types

A union type is allowed to be one of a number of possible types. A union is created with the pipe symbol (`|`)

```
MAX_VALUE: int | float

def spam(chars: str | bytes):
    ...
```

Unions may specify any number of valid types. It is the job of the calling code to tease out the correct type if they must be treated differently.

> Prior to Python 3.9, to specify a union required `typing.Union`:
>
> ```
> from typing import Union
>
> def destroy(junk: Union[Car,Refrigerator,ACUnit]) -&gt None:
>     ...
> ```


Example

hints_union.py

```
class Car:
    def send_to_crusher(self):
        print("Sending car to crusher")

class Refrigerator:
    def remove_doors(self):
        print("removing doors from refrigerator")

class ACUnit:
    def drain_freon(self):
        print("Draining freon from AC Unit")

class Bicycle:
    def remove_parts(self):
        print("removing parts from Bicycle")

def destroy(junk: Car | Refrigerator | ACUnit) -> None:
    if isinstance(junk, Car):
        junk.send_to_crusher()
    elif isinstance(junk, Refrigerator):
        junk.remove_doors()
    elif isinstance(junk, ACUnit):
        junk.drain_freon()

r = Refrigerator()
destroy(r)

c = Car()
destroy(c)

a = ACUnit()
destroy(a)

b = Bicycle()
destroy(b)

destroy("Bicycle")
```

python -m mypy hints_union.py

```
hints_union.py:35: error: Argument 1 to "destroy" has incompatible type "Bicycle";
expected "Car | Refrigerator | ACUnit"  [arg-type]
hints_union.py:37: error: Argument 1 to "destroy" has incompatible type "str"; expected
"Car | Refrigerator | ACUnit"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```


Optional Types

A common specific case of union types is the *optional* type. An optional type is one that is either `None`, or a specified type.

```
def get_record(id: int) -> Record | None:
    row = db.query('WHERE id = ?', id)
    if not row:
        return None
    else:
        return row
```

With Python's support for exceptions, this may seem unusual. But, there are cases where a value may be present, or absent. The `Optional` type is excellent for type-checking these cases, as `mypy` will detect if the wrapped type is being used without a branch for checking the `None` possibility.

This also allows for dealing with detectable default values in a type-safe way.

Example

hints_optional.py

```
def annoy_cat(times: int | None) -> str:
    # This line generates the mypy output:
    # 'note: Right operand is of type "int | None"'
    return 'meow' * times

print(f"{annoy_cat(3) = }")

def print_times(phrase: str, times: int | None = None) -> None:
    """print the phrase some number of times, unless the number is not specified
    """
    if times is None:
        print(phrase)
    else:
        print(phrase * times)

print_times("spam", 5)
print_times("toast")
```

hints_optional.py

```
annoy_cat(3) = 'meowmeowmeow'
spamspamspamspamspam
toast
```

mypy hints_optional.py

```
hints_optional.py:4: error: Unsupported operand types for * ("str" and "None")
[operator]
hints_optional.py:4: note: Right operand is of type "int | None"
Found 1 error in 1 file (checked 1 source file)
```

Prior to Python 3.9, to specify an optional type required `typing.Optional`:

```
from typing import Optional

def get_record(id: int) -> Optional[Record]:
    ...
```


The `typing` Module

The `typing` module makes it easier to refer to containers as a kind of type in Python code. It also contains some special types, such as `Any` and `Generator`.

As of Python 3.9, many types defined in `typing` are no longer needed, as the type itself can be used:

```
from typing import List

a = List[str] # same as list[str]
b = list[str] # typing not needed
```

The `typing` module makes available many *type-wrapper* classes.

The `mypy` documentation includes a cheat sheet at https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html.

See https://docs.python.org/3/library/typing.html for the complete list and full documentation.

Hints for function parameters

Most parameters should not be of type `List`, but rather how the parameter is used, so either an `Iterable` or a `Collection`. Most of the more-specific containers (such as `tuple`, `list`, `dict`, and `set`) should generally only be used as return types.

Why use `Collection` instead of `Iterable`? `Collection`s may be indexed, unlike iterators (AKA generators), which are also included in `Iterable`.

```
from typing import Iterable

def normalize(words: Iterable[str]) -> list[str]:
    return [word.lower().strip() for word in words]
```

Hints for generic parameters and return values

Some functions can accept arguments of any type. To provide a hint for such an argument, use `typing.Any`. A variable annoted with `Any` can be assigned to a variable with any annotation type.

```
from typing import Any

s: str
a: Any = "abc"
s = a  # OK

def normalize(obj: Any) -> str:
    if isinstance(obj, str):
        return obj.strip().lower()
    else:
        return str(obj)

def double(obj: Any) -> Any:
    return 2 * obj
```

Hints for generators

The `typing.Generator` type takes exactly three types for its template: the type yielded, the type sent, and the type returned by the generator. If any of those types are not used, they should be set to `None`.

```
from typing import Iterable, Generator

def normalize(words: Iterable[str]) -> Generator[str, None, None]:
    return (word.lower().strip() for word in words)
```

Forward References

Not all types may be available at the time that a given piece of Python code is compiled to bytecode. In other words, **forward references**, where a type is referred to before it is defined, are needed.

The standard way to do this in Python is by using strings; static analysis tools are expected to handle this forward reference.

```
class First:
    ...
    # The type Second is not yet available
    # to python, so it must be
    # forward-declared using a string
    def process(self, item: 'Second') -> str:
        ...

class Second:
    ...
    # The type First is available to
    # python, so it can just reference
    # the First symbol directly
    def create(self, data: First) -> str:
        ...
```

Other languages use similar concepts for declaring a type without defining it. The `mypy` tool deals correctly with these forward references.

Forward references are also how various operator overloads may need to be written, to refer to the current class.

```
class Matrix:
    def __matmul__(self, obj: 'Matrix') -> 'Matrix':
        ...
```

Forward references can also just be part of a type hint, they need not "gobble up" the entire type hint.

```
class Tree:
    def leaves(self) -> List['Tree']:
        ...
```

Chapter 2 Exercises

Exercise 2-1 (babyname.py)

In the module `babyname` (in the root folder of the labs files), add type hinting to all the methods in the BabyName class.

Try passing incorrect values to the constructor, storing a BabyName object in a variable that is not correctly annotated, or returning an incorrect type from the `add()` method.

Chapter 3: Introduction to Pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with **NumPy** and **MatPlotLib**

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with **NumPy**, **SciPy**, **Matplotlib**, and other scientific packages.

While pandas can handle three (or higher) dimensional data via the `MultiIndex` (hierarchical data) feature, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful split-apply-combine operations, merging, subsetting, and easy-access to plotting functions. It is easy to emulate R's `plyr` package via pandas.

Here are some links that compare Pandas features to the equivalents in R:

- https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_r.html
- https://towardsdatascience.com/cheat-sheet-for-python-dataframe-r-dataframe-syntax-conversions-450f656b44ca
- https://heads0rtai1s.github.io/2020/11/05/r-python-dplyr-pandas/

pandas gets its name from *pan*el *da*ta *s*ystem

Tidy data

- Tidy data is neatly grouped
- Data
  - *Value* = "observation"
  - *Column* = "variable"
  - *Row* = "related observations"
- Pandas best with tidy data

A dataset contains *values*. Those values can be either numbers or strings. Values are grouped into *variables*, which are usually represented as *columns*. For instance, a column might contain "unit price" or "percentage of NaCL". A group of related values is called an *observation*. A *row* represents an observation. Every combination of row and column is a single value.

When data is arranged this way, it is said to be "tidy". Pandas is designed to work best with tidy data.

For instance,

```
Product    SalesYTD
oranges    5000
bananas    1000
grapefruit 10000
```

is tidy data. The variables are "Product" and "SalesYTD", and the observations are the names of the fruits and the sales figures.

The following dataset is NOT tidy:

```
Fruit     oranges bananas grapefruit
SalesYTD  5000    1000    10000
```

To make selecting data easy, Pandas dataframes always have variable labels (columns) and observation labels (row indexes). A row index could be something simple like increasing integers, but it could also be a time series, or any set of strings, including a column pulled from the data set.

variables could be called "features" and observations could be called "samples"

See https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html for a detailed discussion of tidy data.

pandas architecture

- Two main data structures
  - Series – one-dimensional
  - DataFrame – two-dimensional

The two main data structures in pandas are the `Series` and the `DataFrame`. A Series is a one-dimensional indexed list of values, something like a dictionary. A DataFrame is is a two-dimensional grid, with both row and column indexes (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indexes, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

> pandas used to support the `Panel` type, which is more more or less a collection of DataFrames, but Panel has been deprecated in favor of MultiIndex, which provides hierarchical indexing.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- indexes are not positional

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indexes as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. pandas will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_series.py

```
from numpy.random import default_rng
import pandas as pd

NUM_DATA_POINTS = 10
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

rng = default_rng()
data = rng.standard_normal(NUM_DATA_POINTS)

s1 = pd.Series(data, index=index)  # create series with specified index
s2 = pd.Series(data)  # create series with auto-generated index (0, 1, 2, 3, ...)

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n")  # select items from series

print(s1[['a', 'b', 'c']], "\n")  # select items from series

print("slice of elements")
print(s1['b':'d'], "\n")  # select slice of elements

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n")  # get stats on series

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n")  # get stats on series

print('a' in s1)  # test for existence of label
print('m' in s1)  # test for existence of label
print()

s3 = s1 * 10  # create new series with every element of s1 multiplied by 10
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 0:")
print(s3 > 0, "\n")  # create boolean mask from series

print("assign -1 where mask is true")
```

```
s3[s3 < 5] = -1  # set element to -1 where mask is True
print(s3, "\n")

s4 = pd.Series([-0.204708, 0.478943, -0.519439])  # create new series
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n')  # print stats

s = pd.Series([5, 10, 15], ['a', 'b', 'c'])  # create new series with index
print("creating series with index")
print(s)
```

pandas_series.py

```
s1: a   -0.897712
b   -0.453219
c    0.073413
d   -0.648245
e   -0.885363
f   -0.302665
g   -0.247461
h    0.054621
i    1.570261
j   -1.949516
dtype: float64

s2: 0   -0.897712
1   -0.453219
2    0.073413
3   -0.648245
4   -0.885363
5   -0.302665
6   -0.247461
7    0.054621
8    1.570261
9   -1.949516
dtype: float64

selecting elements
h    0.054621
b   -0.453219
dtype: float64

a   -0.897712
b   -0.453219
c    0.073413
dtype: float64

slice of elements
b   -0.453219
c    0.073413
d   -0.648245
dtype: float64

sum(), mean(), min(), max():
-3.685884490592941 -0.3685884490592941 -1.9495157476571314 1.5702607439784804

cumsum(), cumprod():
a   -0.897712
```



```
b   -1.350931
c   -1.277518
d   -1.925763
e   -2.811126
f   -3.113790
g   -3.361251
h   -3.306629
i   -1.736369
j   -3.685884
dtype: float64 a   -0.897712
b    0.406860
c    0.029869
d   -0.019362
e    0.017143
f   -0.005188
g    0.001284
h    0.000070
i    0.000110
j   -0.000215
dtype: float64

True
False

s3 (which is s1 * 10)
a    -8.977125
b    -4.532187
c     0.734131
d    -6.482447
e    -8.853632
f    -3.026646
g    -2.474605
h     0.546215
i    15.702607
j   -19.495157
dtype: float64

boolean mask where s3 > 0:
a    False
b    False
c     True
d    False
e    False
f    False
g    False
h     True
i     True
j    False
```

```
dtype: bool

assign -1 where mask is true
a    -1.000000
b    -1.000000
c    -1.000000
d    -1.000000
e    -1.000000
f    -1.000000
g    -1.000000
h    -1.000000
i    15.702607
j    -1.000000
dtype: float64

s4.max(), .min(), etc.
0.478943 -0.519439 0.998382

creating series with index
a     5
b    10
c    15
dtype: int64
```

DataFrames

- Two-dimensional grid of values
- Row and column labels (indexes)
- Rich set of methods
- Powerful indexing

A DataFrame is the workhorse of pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to initialize a DataFrame from various Python data structures, but most of the time you will be reading data from a file.

Dataframes can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

The panda DataFrame is modeled after R's `data.frame`

Example

pandas_simple_dataframe.py

```
import pandas as pd
from printheader import print_header


columns = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column names
rows = ['a', 'b', 'c', 'd', 'e', 'f']  # row names

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('columns')
print(columns, '\n')

print_header('rows')
print(rows, '\n')

print_header('values')
print(values, '\n')

df = pd.DataFrame(values, index=rows, columns=columns)  # create dataframe with row and
column names
print_header('DataFrame df')
print(df, '\n')
```


pandas_simple_dataframe.py

```
==================================================
=                    columns                     =
==================================================
['alpha', 'beta', 'gamma', 'delta', 'epsilon']

==================================================
=                      rows                      =
==================================================
['a', 'b', 'c', 'd', 'e', 'f']

==================================================
=                     values                     =
==================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640
```

Reading Data

- Many data formats supported
- Creates column indexes from headings
- Auto-creates indexes as needed
- Can specify column for row index

Pandas supports many different input formats. It will read file headings and use them to create column indexes. You can specify a column to use as the row index.

You can provide a list of row or column index values. The length of the index values must match the number of rows or number of columns.

If no indexes are provided, pandas will generate indexes as successive integers starting with 0. (0, 1, 2, 3, …)

The `read_…()` functions have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

`read_csv()` is probably the most frequently used function, and has many options. `read_table()` can be used to read generic flat-file formats.

There are corresponding `to_…()` functions for most of the read functions. `to_csv()` and `to_ndarray()` are very useful.

> See `PandasInputDemo` in the **NOTEBOOKS** folder for examples of reading most types of input.
>
> See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html?highlight=output#io-html for details on the I/O functions.

Example

pandas_read_csv.py

```
import pandas as pd

df = pd.read_csv('../DATA/sales_records.csv')  # Read CSV data into dataframe. Pandas
automatically uses the first row as column names

print(df.describe())  # Get statistics on the numeric columns (use
`df.describe(include='O')` for text columns)
print()

print(df.info())  # Get information on all the columns ('object' means text/string)
print()

print(df.head(5))  # Display first 5 rows of the dataframe (`df.describe(__n__)` displays
n rows)

df['total_sales'] = df['Units Sold'] * df['Unit Price']
print(df)

print(df.info())
print(df.describe())
```

pandas_read_csv.py

```
           Order ID   Units Sold   Unit Price    Unit Cost
count  5.000000e+03  5000.000000  5000.000000  5000.000000
mean   5.486447e+08  5030.698200   265.745564   187.494144
std    2.594671e+08  2914.515427   218.716695   176.416280
min    1.000909e+08     2.000000     9.330000     6.920000
25%    3.201042e+08  2453.000000    81.730000    35.840000
50%    5.523150e+08  5123.000000   154.060000    97.440000
75%    7.687709e+08  7576.250000   437.200000   263.330000
max    9.998797e+08  9999.000000   668.270000   524.960000

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Region          5000 non-null   object
 1   Country         5000 non-null   object
 2   Item Type       5000 non-null   object
 3   Sales Channel   5000 non-null   object
```

```
 4   Order Priority  5000 non-null   object 
 5   Order Date      5000 non-null   object 
 6   Order ID        5000 non-null   int64  
 7   Ship Date       5000 non-null   object 
 8   Units Sold      5000 non-null   int64  
 9   Unit Price      5000 non-null   float64
 10  Unit Cost       5000 non-null   float64
dtypes: float64(2), int64(2), object(7)
memory usage: 429.8+ KB
None

                              Region  ... Unit Cost
0  Central America and the Caribbean  ...    159.42
1  Central America and the Caribbean  ...     97.44
2                             Europe  ...     31.79
3                               Asia  ...    117.11
4                               Asia  ...     97.44

[5 rows x 11 columns]
                                 Region  ... total_sales
0     Central America and the Caribbean  ...   140914.56
1     Central America and the Caribbean  ...   330640.86
2                                Europe  ...   226716.10
3                                  Asia  ...  1854591.20
4                                  Asia  ...  1150758.36
...                                 ...  ...         ...
4995              Australia and Oceania  ...  3545172.35
4996       Middle East and North Africa  ...   117694.56
4997                               Asia  ...  1328477.12
4998                             Europe  ...  1028324.80
4999                 Sub-Saharan Africa  ...   377447.00

[5000 rows x 12 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Region          5000 non-null   object 
 1   Country         5000 non-null   object 
 2   Item Type       5000 non-null   object 
 3   Sales Channel   5000 non-null   object 
 4   Order Priority  5000 non-null   object 
 5   Order Date      5000 non-null   object 
 6   Order ID        5000 non-null   int64  
 7   Ship Date       5000 non-null   object 
 8   Units Sold      5000 non-null   int64  
 9   Unit Price      5000 non-null   float64
```



```
 10  Unit Cost       5000 non-null   float64
 11  total_sales     5000 non-null   float64
dtypes: float64(3), int64(2), object(7)
memory usage: 468.9+ KB
None
           Order ID   Units Sold   Unit Price    Unit Cost   total_sales
count  5.000000e+03  5000.000000  5000.000000  5000.000000  5.000000e+03
mean   5.486447e+08  5030.698200   265.745564   187.494144  1.325738e+06
std    2.594671e+08  2914.515427   218.716695   176.416280  1.475375e+06
min    1.000909e+08     2.000000     9.330000     6.920000  6.531000e+01
25%    3.201042e+08  2453.000000    81.730000    35.840000  2.574168e+05
50%    5.523150e+08  5123.000000   154.060000    97.440000  7.794095e+05
75%    7.687709e+08  7576.250000   437.200000   263.330000  1.839975e+06
max    9.998797e+08  9999.000000   668.270000   524.960000  6.672676e+06
```

Table 2. pandas I/O functions

Format	Input function	Output function
CSV	`read_csv()`	`to_csv()`
Delimited file (generic)	`read_table()`	`to_csv()`
Excel worksheet	`read_excel()`	`to_excel()`
File with fixed-width fields	`read_fwf()`	
Google BigQuery	`read_gbq()`	`to_gbq()`
HDF5	`read_hdf()`	`to_hdf()`
HTML table	`read_html()`	`to_html()`
JSON	`read_json()`	`to_json()`
OS clipboard data	`read_clipboard()`	`to_clipboard()`
Parquet	`read_parquet()`	`to_parquet()`
pickle	`read_pickle()`	`to_pickle()`
SAS	`read_sas()`	
SQL query	`read_sql()`	`to_sql()`

All **read_...()** functions return a new **DataFrame**, except **read_html()**, which returns a list of **DataFrames**

Data summaries

- `describe()` *statistical details of numeric columns*
- `describe(include="O")` details of **object** columns (strings)
- `info()` *per-column details (shallow memory use)*
- `info(memory_usage='deep')` *actual memory use*

You can call the `describe()` and `info()` methods on a dataframe to get summaries of the kind of data contained.

The `describe()` method, by default, shows statistics on all numeric columns. Add `include='int'` or `include='float'` to restrict the output to those types. `include='all'` will show all types, including "objects" (AKA text).

To show just objects (strings), use `include='O'`. This will show all text columns. You can compare the **count** and **unique** values to check the *cardinality* of the column, or how many distinct values there are. Columns with few unique values are said to have low cardinality, and are candidates for saving space by using the `Category` data type.

The `info()` method will show the names and types of each column, as well as the count of non-null values. Adding `memory_usage='deep'` will display the total memory actually used by the dataframe. (Otherwise, it’s only the memory used by the top-level data structures).

Example

pandas_data_summaries.py

```
import pandas as pd
from printheader import print_header

df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',', index_col=1)

print_header('df.head()')
print(df.head())
print()

print_header('df.describe()')
print(df.describe())

print_header("df.describe(include='int')")
print(df.describe(include='int'))

print_header("df.describe(include='all')")
print(df.describe(include='all'))

print_header("df.info()")
print(df.info())
```

pandas_data_summaries.py

```
==================================================
=                  df.head()                     =
==================================================
                                                Airport  ...  Percent change 2010-2011
Code                                                     ...
ATL   Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                     -22.6
ORD          Chicago, IL (Chicago O'Hare International)  ...                     -25.5
DFW        Dallas, TX (Dallas/Fort Worth International)  ...                     -23.7
DEN                   Denver, CO (Denver International)  ...                     -23.1
LAX         Los Angeles, CA (Los Angeles International)  ...                     -19.6

[5 rows x 9 columns]

==================================================
=                df.describe()                   =
==================================================
       2001 Rank  ...  Percent change 2010-2011
count  50.000000  ...                 50.000000
mean   26.460000  ...                -23.758000
```



```
std    15.761242  ...                2.435963
min     1.000000  ...              -32.200000
25%    13.250000  ...              -25.275000
50%    26.500000  ...              -23.650000
75%    38.750000  ...              -22.075000
max    59.000000  ...              -19.500000

[8 rows x 8 columns]
==================================================
=           df.describe(include='int')           =
==================================================
       2001 Rank    2001 Total  ...  2011 Rank         Total
count  50.000000  5.000000e+01  ...   50.00000  5.000000e+01
mean   26.460000  9.848488e+06  ...   25.50000  8.558513e+06
std    15.761242  7.042127e+06  ...   14.57738  6.348691e+06
min     1.000000  2.503843e+06  ...    1.00000  2.750105e+06
25%    13.250000  4.708718e+06  ...   13.25000  3.300611e+06
50%    26.500000  7.626439e+06  ...   25.50000  6.716353e+06
75%    38.750000  1.282468e+07  ...   37.75000  1.195822e+07
max    59.000000  3.638426e+07  ...   50.00000  3.303479e+07

[8 rows x 6 columns]
==================================================
=           df.describe(include='all')           =
==================================================
                                                  Airport  ...  Percent change 2010-2011
count                                                  50  ...                 50.000000
unique                                                 50  ...                       NaN
top     Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                       NaN
freq                                                    1  ...                       NaN
mean                                                  NaN  ...                -23.758000
std                                                   NaN  ...                  2.435963
min                                                   NaN  ...                -32.200000
25%                                                   NaN  ...                -25.275000
50%                                                   NaN  ...                -23.650000
75%                                                   NaN  ...                -22.075000
max                                                   NaN  ...                -19.500000

[11 rows x 9 columns]
==================================================
=                   df.info()                    =
==================================================
<class 'pandas.core.frame.DataFrame'>
Index: 50 entries, ATL to IND
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Airport                   50 non-null     object
```

```
 1   2001 Rank                  50 non-null     int64
 2   2001 Total                 50 non-null     int64
 3   2010 Rank                  50 non-null     int64
 4   2010 Total                 50 non-null     int64
 5   2011 Rank                  50 non-null     int64
 6    Total                     50 non-null     int64
 7   Percent change 2001-2011  50 non-null     float64
 8   Percent change 2010-2011  50 non-null     float64
dtypes: float64(2), int64(6), object(1)
memory usage: 3.9+ KB
None
```


Selecting rows and columns

- Similar to normal Python or **numpy**
- Uses `[]` (getitem operator)
  - Strings or iterables select columns
  - Slices select rows

One of the real strengths of pandas is the ability to easily select desired rows and columns. This can be done with simple subscripting using the `[]` (getitem) operator.

For selecting one column, use the column name as the subscript value. This selects the entire column. To select multiple columns, use an iterable of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, `dataframe[x:y]` selects rows x through y, but `dataframe[x]` selects column x.

To select a single row, use a slice with the same start and stop value.

Example

pandas_selecting.py

```
import pandas as pd
from printheader import print_header

columns = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
index = ['a', 'b', 'c', 'd', 'e', 'f']  # row labels

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=columns)  # create dataframe with data,
row labels, and column labels
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n')  # select column 'alpha' -- single value selects column by name

print_header("df.beta")
print(df.beta, '\n')  # same, but alternate syntax (only works if column name is letters,
digits, and underscores)

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']])  # select columns -- note index is an iterable
print()

print_header("df['b':'e']")
print(df['b':'e'], '\n')  # select rows 'b' through 'e' using slice of row labels

print_header("df['b':'b']")
print(df['b':'b'], '\n')  # select row 'b' only using slice of row labels (returns
dataframe)

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e'])  # select columns AND slice rows
print()
```


pandas_selecting.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                  df['alpha']                   =
==================================================
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha, dtype: int64

==================================================
=                    df.beta                     =
==================================================
a    110
b    210
c    310
d    410
e    510
f    610
Name: beta, dtype: int64

==================================================
=         df[['alpha','epsilon','beta']]         =
==================================================
   alpha  epsilon  beta
a    100      140   110
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
f    600      640   610

==================================================
```

```
=                  df['b':'e']                   =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540

==================================================
=                  df['b':'b']                   =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240

==================================================
=    df[['alpha','epsilon','beta']]['b':'e']     =
==================================================
   alpha  epsilon  beta
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
```


Indexing with `.loc` and `.at`

- `loc[row-spec,col-spec]`
- `at[row, col]`
- row or column specs
  - single name
  - iterable of names
  - range (inclusive) of names
- `.at[row, col]` single value

The `.loc` indexer provides more consistent selecting of rows and columns. `.loc` uses row and column *names* (indexes).

`.loc` uses the *getitem* operator `[]`, with the syntax `[row-specifier, column-specifier]`.

The row or column specifier can be either a single name, an iterable of names, or a range of names. The end of a name index range is inclusive.

The `.at[]` indexer can be used to select a single value at a given row and column: `df.at[47, "color"]`.

To select all rows, use `:`. To select all columns, omit the column specifier.

Example

pandas_loc.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.loc['b', 'delta']")  # one value
print(df.loc['b', 'delta'], "\n")


print_header("df.loc['b']")  # one row
print(df.loc['b'], '\n')

print_header("df.loc[:,'delta']")  # one column
print(df.loc[:,'delta'], '\n')


print_header("df.loc['b': 'd']")  # range of rows
print(df.loc['b':'d', :], '\n')
print(df.loc['b':'d'], '\n')   # shorter version

print_header("df.loc[:,'beta':'delta'")  # range of columns
print(df.loc[:, 'beta':'delta'], "\n")

print_header("df.loc['b':'d', 'beta':'delta']")  # ranges of rows and columns
print(df.loc['b':'d', 'beta':'delta'], '\n')

print_header("df.loc[['b', 'e', 'a']]")  # iterable of rows
print(df.loc[['b', 'e', 'a']], "\n")
```



```
print_header("df.loc[:, ['gamma', 'alpha', 'epsilon']]")  # iterable of columns
print(df.loc[:, ['gamma', 'alpha', 'epsilon']], "\n")

print_header("df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]")  # iterables of
rows and columns
print(df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']], "\n")
```

pandas_loc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=              df.loc['b', 'delta']              =
==================================================
230

==================================================
=                  df.loc['b']                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=               df.loc[:,'delta']                =
==================================================
a    130
b    230
c    330
d    430
e    530
f    630
Name: delta, dtype: int64
```

```
==================================================
=              df.loc['b': 'd']                  =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

==================================================
=          df.loc[:,'beta':'delta'               =
==================================================
   beta  gamma  delta
a   110    120    130
b   210    220    230
c   310    320    330
d   410    420    430
e   510    520    530
f   610    620    630

==================================================
=      df.loc['b':'d', 'beta':'delta']           =
==================================================
   beta  gamma  delta
b   210    220    230
c   310    320    330
d   410    420    430

==================================================
=          df.loc[['b', 'e', 'a']]               =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=    df.loc[:, ['gamma', 'alpha', 'epsilon']]    =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
```



```
d    420    400      440
e    520    500      540
f    620    600      640

==================================================
df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140
```

Indexing with .iloc and .iat

- `.iloc[row-spec,col-spec]` for 0-based position (integers only)
- row or column specs
  - single positional index
  - iterable of indexes
  - range (exclusive) of integers
- `.iat[row,col]` for single value

The `.iloc` indexer provides access to rows and columns using the *position*. Indexers are integers, starting at 0. Like `.loc`, it uses the *getitem* operator `[]`, with the syntax `[row-specifier, column-specifier]`.

For `.iloc[]`, the specifier can be either a single positional index (0-based), iterable of indexes, or a range of indexes. The end of a positional index range is exclusive.

To select all rows use `:`. To select all columns, omit the column specifier.

Use `.iat[]` to select a single value by position.

Example

pandas_iloc.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.iloc[1, 3]")  # one value
print(df.iloc[1, 3], "\n")


print_header("df.iloc[1]")  # one row
print(df.iloc[1], '\n')

print_header("df.iloc[:,3]")  # one column
print(df.iloc[:, 3], '\n')


print_header("df.iloc[1: 3]")  # range of rows
print(df.iloc[1:3, :], '\n')
print(df.iloc[1:3], '\n')   # shorter version

print_header("df.iloc[:,1:3]")  # range of columns
print(df.iloc[:, 1:3], "\n")

print_header("df.iloc[1:3, 1:3]")  # ranges of rows and columns
print(df.iloc[1:3, 1:3], '\n')

print_header("df.iloc[[1, 4, 0]]")  # iterable of rows
print(df.iloc[[1, 4, 0]], "\n")
```

```
print_header("df.iloc[:, [2, 0, 4]]")  # iterable of columns
print(df.iloc[:, [2, 0, 4]], "\n")

print_header("df.iloc[[1, 4, 0], [2, 0, 4]]")  # iterables of rows and columns
print(df.iloc[[1, 4, 0], [2, 0, 4]], "\n")

print_header("df.iat[2, 3]")
print(df.iat[2, 3], "\n")
```

pandas_iloc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                 df.iloc[1, 3]                  =
==================================================
230

==================================================
=                   df.iloc[1]                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=                  df.iloc[:,3]                  =
==================================================
a    130
b    230
c    330
d    430
e    530
```

```
f    630
Name: delta, dtype: int64

==================================================
=                 df.iloc[1: 3]                  =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

==================================================
=                 df.iloc[:,1:3]                 =
==================================================
   beta  gamma
a   110    120
b   210    220
c   310    320
d   410    420
e   510    520
f   610    620

==================================================
=               df.iloc[1:3, 1:3]                =
==================================================
   beta  gamma
b   210    220
c   310    320

==================================================
=               df.iloc[[1, 4, 0]]               =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=             df.iloc[:, [2, 0, 4]]              =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
d    420    400      440
```

```
e    520    500      540
f    620    600      640

==================================================
=         df.iloc[[1, 4, 0], [2, 0, 4]]          =
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140

==================================================
=                  df.iat[2, 3]                  =
==================================================
330
```


Broadcasting

- Operation is applied across rows and columns
- Can be restricted to selected rows and columns
- Sometimes called vectorization
- Use apply() for more complex operations

An operator or function can be applied to an entire dataframe, or a selected portion. This is more efficient than iterating over the rows and columns. This is called 'broadcasting'.

For instance, if you multiply a numeric column by 10, every value in that column will be multipled by 10. This works for most builtin operators.

User-defined functions can also be broadcast across rows and columns. The function should take one argument and return one value.

For more complex operations, the apply() method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example

pandas_broadcasting.py

```
import pandas as pd
from printheader import print_header

columns = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
rows = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D')  # date range to be used
as row indexes

values = [  # sample data
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

df = pd.DataFrame(values, rows, columns)  # create dataframe from data
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple all values")
print(df * 3)
print()  # multiply every value by 3

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5  # multiply values in column 'gamma' by 1.
print(df)
print()

def square_root(n):
    return n ** .5

df['alpha'] = square_root(df['alpha'])
print_header("Apply square_root() to column alpha")
print(df, '\n')
```


pandas_broadcasting.py

```
==================================================
=               Basic DataFrame:                 =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    120    930      140
2013-01-02    250   210    120    130      840
2013-01-03    300   310    520    430      340
2013-01-04    275   410    420    330      777
2013-01-05    300   510    120    730      540
2013-01-06    150   610    320    690      640

==================================================
=              Triple all values                 =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    300   330    360   2790      420
2013-01-02    750   630    360    390     2520
2013-01-03    900   930   1560   1290     1020
2013-01-04    825  1230   1260    990     2331
2013-01-05    900  1530    360   2190     1620
2013-01-06    450  1830    960   2070     1920

==================================================
=         Multiply column gamma by 1.5           =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110  180.0    930      140
2013-01-02    250   210  180.0    130      840
2013-01-03    300   310  780.0    430      340
2013-01-04    275   410  630.0    330      777
2013-01-05    300   510  180.0    730      540
2013-01-06    150   610  480.0    690      640

==================================================
=     Apply square_root() to column alpha        =
==================================================
                alpha  beta  gamma  delta  epsilon
2013-01-01  10.000000   110  180.0    930      140
2013-01-02  15.811388   210  180.0    130      840
2013-01-03  17.320508   310  780.0    430      340
2013-01-04  16.583124   410  630.0    330      777
2013-01-05  17.320508   510  180.0    730      540
2013-01-06  12.247449   610  480.0    690      640
```

Counting unique occurrences

- Use `.value_counts()`
- Called from column

To count the unique occurrences within a column, call the method `value_counts()` on the column. It returns a `Series` object with the column values and their counts.

Example

pandas_unique.py

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_excel(
    'https://qrc.depaul.edu/Excel_Files/Presidents.xlsx',
    index_col="No",  # use term as row index
    sheet_name='Master',  # name of worksheet
    na_values='NA()')  # use NA() for missing values

print("First 5 rows")
print(df.head(), '\n')

print("First row")
print(df.loc[1], '\n')

party_counts = df['Political Party'].value_counts()
print("Party counts")
print(party_counts)

# plot the data
plt.figure(figsize=(20.0,8.0))  # set figure size
party_counts.plot(kind='barh')  # plot a horizontal bar graph
plt.savefig("parties.png")    # save graph to file
# plt.show()  # uncomment to display graph
```

pandas_unique.py

```
First 5 rows
            President  Years in office  ...  % electoral  % popular
No                                      ...
1   George Washington              8.0  ...   100.000000        NaN
2          John Adams              4.0  ...    94.964029        NaN
3    Thomas Jefferson              8.0  ...    53.284672        NaN
4       James Madison              8.0  ...    69.318182        NaN
5        James Monroe              8.0  ...    82.805430        NaN

[5 rows x 15 columns]

First row
President                  George Washington
Years in office                          8.0
Year first inaugurated                  1789
Age at inauguration                       57
State elected from                  Virginia
# of electoral votes                    69.0
# of popular votes                       NaN
National total votes                     NaN
Total electoral votes                   69.0
Rating points                          842.0
Political Party                          NaN
Occupation                           Planter
College                                  NaN
% electoral                            100.0
% popular                                NaN
Name: 1, dtype: object

Party counts
Political Party
Republican              19
Democrat                16
Whig                     4
Democratic-Republican    4
Federalist               1
National Union           1
Name: count, dtype: int64
```

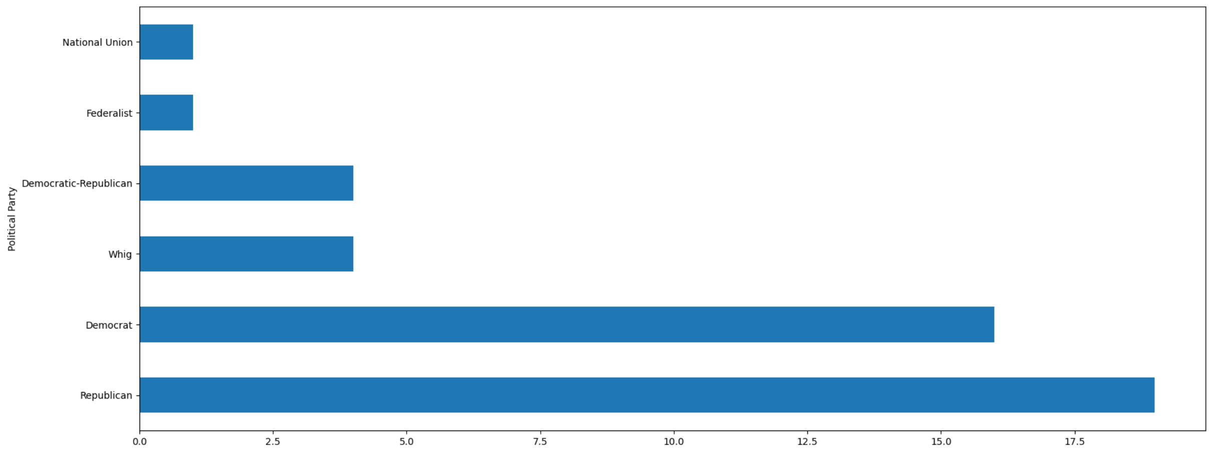



Figure 1. Bar graph of value counts

Creating new columns

- Assign iterable to new column name
- Length of iterable must match number of rows
- Can use operators or functions on existing columns
- Single value is replicated

For simple cases, it's easy to create new columns. Just assign an iterable to a new column name. The length of the iterable must match the number of rows. The column will be appended at the end of the exiting columns.

One way to do this is to combine other columns with an operator or function.

Assigning a single value will replicate the value across all rows.

To insert a column at a specified position, use `dataframe.insert(name, pos, data)`

Example

pandas_new_columns.py

```
import pandas as pd

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols)

def times_ten(x):
    return x * 10

df['zeta'] = df['delta'] * df['epsilon'] # product of two columns
df['eta'] = times_ten(df.alpha) # user-defined function
df['theta'] = df.sum(axis=1)  # sum each row
df['iota'] = df.mean(axis=1)  # avg of each row
df['kappa'] = df.loc[:,'alpha':'epsilon'].mean(axis=1)
# column kappa is avg of selected columns

# assign any iterable with same length as number of rows
animals = ['wombat', 'honey badger', 'platypus', 'coatimundi', 'fennec fox', 'naked mole
rat']
df['lambda'] = animals

# single value is replicated across rows
df['mu'] = 5

# insert column at specified position
values = [10 * n for n in range(1, len(df) + 1)]
df.insert(0, "omega", values)

print(df)
```

pandas_new_columns.py

```
   omega  alpha  beta  gamma  ...      iota  kappa          lambda  mu
a     10    100   110    120  ...    4950.0  120.0          wombat   5
b     20    200   210    220  ...   14575.0  220.0    honey badger   5
c     30    300   310    320  ...   29200.0  320.0        platypus   5
d     40    400   410    420  ...   48825.0  420.0      coatimundi   5
e     50    500   510    520  ...   73450.0  520.0      fennec fox   5
f     60    600   610    620  ...  103075.0  620.0  naked mole rat   5

[6 rows x 13 columns]
```

Removing entries

- Remove specified rows or columns
- Remove rows with invalid data.

To remove columns or rows by index, use the `.drop()` method. To remove rows or columns with invalid data, use `.dropna()`. These methods return a new dataframe with the rows or columns removed.

Use `axis=1` to drop columns, or `axis=0` to drop rows. The default value for `axis` is 0.

Both methods support the `inplace` argument to remove data from the dataframe itself rather than returning a new dataframe.

Example

pandas_drop.py

```
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols)  # create dataframe
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1)  # drop columns beta and delta (axes: 0=rows,
1=columns)
print_header("After dropping beta and delta:")
print(df2, '\n')

print_header("After dropping rows b, c, and e")
df3 = df.drop(['b', 'c', 'e'])  # drop rows b, c, and e
print(df3)

print_header(" In-place drop")
df.drop(['beta', 'gamma'], axis=1, inplace=True)
print(df, "\n")

df.drop(['b', 'c'], inplace=True)
print(df)
print('-' * 60)

# dropping N/A values

values2 = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, pd.NA, 420, 430, 440],
    [500, 510, 520, pd.NA, 540],
```



```
    [600, 610, 620, 630, 640],
]

df2 = pd.DataFrame(values2, index=index, columns=cols)  # create dataframe
print_header('DataFrame df2')
print(df2, '\n')

na1 = df2.dropna(axis=1)  # drop columns with N/A
print_header("Dataframe na1")
print(na1, '\n')

na2 = df2.dropna(axis=0)  # drop rows with N/A (default value for axis)
print_header("Dataframe na2")
print(na2, '\n')

df2.dropna(inplace=True, axis=1)
print_header("Dataframe df2")
print(df2, '\n')
```

pandas_drop.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=         After dropping beta and delta:         =
==================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

==================================================
=        After dropping rows b, c, and e         =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
d    400   410    420    430      440
f    600   610    620    630      640
==================================================
=                  In-place drop                 =
==================================================
   alpha  delta  epsilon
a    100    130      140
b    200    230      240
c    300    330      340
d    400    430      440
e    500    530      540
f    600    630      640

   alpha  delta  epsilon
a    100    130      140
d    400    430      440
e    500    530      540
f    600    630      640
```

```
------------------------------------------------------------
==================================================
=                 DataFrame df2                  =
==================================================
   alpha  beta  gamma delta  epsilon
a    100   110    120   130      140
b    200   210    220   230      240
c    300   310    320   330      340
d    400  <NA>    420   430      440
e    500   510    520  <NA>      540
f    600   610    620   630      640

==================================================
=                 Dataframe na1                  =
==================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

==================================================
=                 Dataframe na2                  =
==================================================
   alpha beta  gamma delta  epsilon
a    100  110    120   130      140
b    200  210    220   230      240
c    300  310    320   330      340
f    600  610    620   630      640

==================================================
=                 Dataframe df2                  =
==================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640
```

Useful pandas methods

Table 3. Methods and attributes for fetching DataFrame/Series data

Method	Description
`DF.columns()`	Get or set column labels
`DF.shape()` `S.shape()`	Get or set shape (length of each axis)
`DF.head(n)` `DF.tail(n)`	Return n items (default 5) from beginning or end
`DF.describe()` `S.describe()`	Display statistics for dataframe
`DF.info()`	Display column attributes
`DF.values` `S.values`	Get the actual values from a data structure
`DF.loc[row_indexer`[1]`, col_indexer]`	Multi-axis indexing by label (not by position)
`DF.iloc[row_indexer`[2]`, col_indexer]`	Multi-axis indexing by position (not by labels)

[1] Indexers can be label, slice of labels, or iterable of labels.
[2] Indexers can be numeric index (0-based), slice of indexes, or iterable of indexes.

Table 4. Methods for Computations/Descriptive Stats (called from pandas)

Method	Returns
`abs()`	absolute values
`corr()`	pairwise correlations
`count()`	number of values
`cov()`	Pairwise covariance
`cumsum()`	cumulative sums
`cumprod()`	cumulative products
`cummin()`, `cummax()`	cumulative minimum, maximum
`kurt()`	unbiased kurtosis
`median()`	median
`min()`, `max()`	minimum, maximum values
`prod()`	products
`quantile()`	values at given quantile
`skew()`	unbiased skewness
`std()`	standard deviation
`var()`	variance

these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

More **pandas** ...

At this point, please view the following Jupyter notebooks for more pandas exploration:

- PandasIntro.ipynb
- PandasInputDemo.ipynb
- PandasSelectionDemo.ipynb
- PandasOptions.ipynb
- PandasMerging.ipynb

> The instructor can explain how to start the Jupyterlab server.

Chapter 3 Exercises

Exercise 3-1 (add_columns.py)

Read in the file **sales_records.csv** as shown in the early part of the chapter. Add three new columns to the dataframe:

- Total Revenue (*units sold x unit price*)
- Total Cost (*units sold x unit cost*)
- Total Profit (*total revenue - total cost*)

Exercise 3-2 (parasites.py))

The file parasite_data.csv, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise…). Read parasite_data.csv into a DataFrame. Print out all rows where the Shannon Diversity is >= 1.0.

Chapter 4: Introduction to Matplotlib

Objectives

- Understand what matplotlib can do
- Create many kinds of plots
- Label axes, plots, and design callouts

About matplotlib

- matplotlib is a package for making 2D plots
- Emulates MATLAB®, but not a drop-in replacement
- matplotlib's philosophy: create simple plots simply
- Plots are publication quality
- Plots can be rendered in GUI applications

This chapter's discussion of matplotlib will use the iPython notebook named **MatplotlibExamples.ipynb**. Please start the iPython notebook server and load this notebook, as directed by the instructor.

matplotlib architecture

- pylab/pyplot front end plotting functions
- API create/manage figures, text, plots
- backends device-independent renderers

matplotlib consists of roughly three parts: pylab/pyplot, the API, and and the backends.

pyplot is a set of functions which allow the user to quickly create plots. Pyplot functions are named after similar functions in MATLAB.

The API is a large set of classes that do all the work of creating and manipulating plots, lines, text, figures, and other graphic elements. The API can be called directly for more complex requirements.

pylab combines pyplot with numpy. This makes pylab emulate MATLAB more closely, and thus is good for interactive use, e.g., with iPython. On the other hand, pyplot alone is very convenient for scripting. The main advantage of pylab is that it imports methods from both pyplot and pylab.

There are many backends which render the in-memory representation, created by the API, to a video display or hard-copy format. For example, backends include PS for Postscript, SVG for scalable vector graphics, and PDF.

The normal import is

```
import matplotlib.pyplot as plt
```


Matplotlib Terminology

- Figure
- Axis
- Subplot

A Figure is one "picture". It has a border ("frame"), and other attributes. A Figure can be saved to a file.

A Plot is one set of values graphed onto the Figure. A Figure can contain more than one Plot.

Axes and Subplot are similar; the difference is how they get placed on the figure. Subplots allow multiple plots to be placed in a rectangular grid. Axes allow multiple plots to placed at any location, including within other plots, or overlapping.

matplotlib uses default objects for all of these, which are sufficient for simple plots. You can explicitly create any or all of these objects to fine-tune a graph. Most of the time, for simple plots, you can accept the defaults and get great-looking figures.

Matplotlib Keeps State

- Primary method is matplotlib.pyplot()
- The current figure can have more than one plot
- Calling show() displays the current figure

matplotlib.pyplot is the workhorse of figure drawing. It is usually aliased to "plt".

While Matplotlib is object oriented, and you can manually create figures, axes, subplots, etc., pyplot() will create a figure object for you automatically, and commands called from pyplot() (usually through the **plt** alias) will work on that object.

Calling **plt.plot()** plots one set of data on the current figure. Calling it again adds another plot to the same figure.

plt.show() displays the figure, although iPython may display each separate plot, depending on the current settings.

You can pass one or two datasets to plot(). If there are two datasets, they need to be the same length, and represent the x and y data.

What Else Can You Do?

- Multiple plots
- Control ticks on any axis
- Scatter plots
- Polar axes
- 3D Plots
- Quiver plots
- Pie Charts

There are many other types of drawings that matplotlib can create. Also, there are many more style details that can be tweaked. See http://matplotlib.org/gallery.html for dozens of sample plots and their source.

There are many extensions (AKA toolkits) for Matplotlib, including Seaborne, CartoPy, at Natgrid.

Matplotlib Demo

At this point, please open the notebook **MatPlotLibExamples.ipynb** for an instructor-led tour of MPL features.

Chapter 4 Exercises

Exercise 4-1 (energy_use_plot.py)

Using the file energy_use_quad.csv in the DATA folder, use matplotlib to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don't plot the "as a percent…".

You can do this in iPython, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use pandas to read the data. The columns are, in Python terms:

```
['Desc',"1960","1965","1970","1975","1980","1985","1990","1991","1992","1993","1994","199
5","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006","2007","2
008","2009","2010","2011"]
```

See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Index

