Intermediate Python Programming

None

Version 1.0, September 2025

Table of Contents

About Intermediate Python Programming

Course Outline

Day 1

Day 2

Available topics if time permits

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 **pycirrusinter**. See the setup guides for details.

What’s in the files?

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

The following folders *may* be present:

pycirrusinter/BIG_DATA/ contains large data files used in examples and answers
pycirrusinter/NOTEBOOKS/ Jupyter notebooks for use in class
pycirrusinter/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: Errors and Logging

Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects
- Setting up basic logging
- Logging exceptions

Exceptions

- Generated when runtime errors occur
- Usually fatal if not handled

Even if code is syntactically correct, run-time errors can occur once a script is launched. A common error is to attempt to open a non-existent file. Such errors are also called *exceptions*, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

Python provides `try-except` blocks to catch, or handle, the exceptions. When an exception is caught, you can log the error, provide a default value, or gracefully shut down the program, depending on the severity of the error.

Custom exceptions can be created by sub-classing the Exception object.

Example

exception_unhandled.py

```
x = 5
y = "cheese"

z = x + y  # Adding a string to an int raises TypeError
```

exception_unhandled.py 2>&1

```
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/exception_unhandled.py", line
4, in <module>
    z = x + y  # Adding a string to an int raises TypeError
        ~~^~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code in a `try` block. After the `try` block, you must specify an `except` block with the expected exception. If an exception is raised in the `try` block, execution stops and the interpreter checks to see if the exception matches the `except` block. If it matches, it executes the except block and execution continues; otherwise, the exception is treated as fatal (the default behavior) and the interpreter exits.

Example

exception_simple.py

```
try:  # Execute code that might have a problem
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err:    # Catch the expected error; assign error object to err
    print("Naughty programmer! ", err)

print("After try-except")  # Get here whether or not exception occurred
```

exception_simple.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
After try-except
```

Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

Example

exception_multiple.py

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err:  # Use a tuple of 2 or more exception types
    print("Naughty programmer! ", err)
```

exception_multiple.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Handling generic exceptions

- Use **Exception**
- Specify except with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the try block.

Example

exception_generic.py

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: # Will catch _any_ exception
    print("Naughty programmer! ", err)
```

exception_generic.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Ignoring exceptions

- Use pass
- Not usually recommended

Use the pass statement to do nothing when an exception occurs

Because the except clause must contain some code, the pass statement fulfills the syntax without doing anything.

Example

exception_ignore.py

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): # Catch exceptions, and do nothing
    pass
```

exception_ignore.py

```
no output
```

It is usually a bad idea to completely ignore an error. It might be indicating an unexpected problem in your code.

Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last `except` block can be followed by an `else` block. The code in the `else` block is executed only if there were no exceptions raised in the `try` block. Exceptions in the `else` block are not handled by the preceding `except` blocks.

The `else` block lets you make sure that some code related to the `try` clause (and before the `finally` clause) is only run if there's no exception, without trapping the exception specified in the `except` clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
# we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```

Example

exception_else.py

```
numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print(f"{err}: x = {x} y = {y}")
    else:
        total += quotient  # Only if no exceptions were raised
print(total)
```

exception_else.py

```
division by zero: x = 5 y = 0
5.2
```

Cleaning up with finally

- Code runs whether or not exception raised
    - Even if script exits in `except` block
- Clean up resources

A `finally` block can be used in addition to, or instead of, an `except` block. The code in a `finally` block is executed whether or not an exception occurs. The `finally` block is executed after the `try`, `except`, and `else` blocks.

What makes `finally` different from just putting statements after try-except-else is that the `finally` block will execute even if there is a `return()` or `exit()` in the `except` block.

The purpose of a `finally` block is to clean up any resources left over from the `try` block. Examples include closing network connections and removing temporary files.

Example

exception_finally.py

```
try:
    x = 5
    y = 37
    z = x + y
    print("z is", z)
except TypeError as err:    # Catch TypeError
    print("Caught exception:", err)
finally:
    print("Don't care whether we had an exception")  # Print whether TypeError is caught
or not

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

exception_finally.py

```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```

The Standard Exception Hierarchy (Python 3.11)

```
Exception
     ├──── ArithmeticError
     │     ├──── FloatingPointError
     │     ├──── OverflowError
     │     └──── ZeroDivisionError
     ├──── AssertionError
     ├──── AttributeError
     ├──── BufferError
     ├──── EOFError
     ├──── ExceptionGroup [BaseExceptionGroup]
     ├──── ImportError
     │     └──── ModuleNotFoundError
     ├──── LookupError
     │     ├──── IndexError
     │     └──── KeyError
     ├──── MemoryError
     ├──── NameError
     │     └──── UnboundLocalError
     ├──── OSError
     │     ├──── BlockingIOError
     │     ├──── ChildProcessError
     │     ├──── ConnectionError
     │     │     ├──── BrokenPipeError
     │     │     ├──── ConnectionAbortedError
     │     │     ├──── ConnectionRefusedError
     │     │     └──── ConnectionResetError
     │     ├──── FileExistsError
     │     ├──── FileNotFoundError
     │     ├──── InterruptedError
     │     ├──── IsADirectoryError
     │     ├──── NotADirectoryError
     │     ├──── PermissionError
     │     ├──── ProcessLookupError
     │     └──── TimeoutError
     ├──── ReferenceError
     ├──── RuntimeError
     │     ├──── NotImplementedError
     │     └──── RecursionError
     ├──── StopAsyncIteration
     ├──── StopIteration
     ├──── SyntaxError
     │     └──── IndentationError
     │          └──── TabError
     ├──── SystemError
     ├──── TypeError
     ├──── ValueError
         └──── UnicodeError
             ├──── UnicodeDecodeError
             ├──── UnicodeEncodeError
             └──── UnicodeTranslateError
```

Simple Logging

- Configure with `logging.basicConfig()`
    - Specify file name
    - Configure the minimum logging level
- Messages added at different levels
- Call methods on `logging`

For simple logging, just configure the log file name and minimum logging level with the `basicConfig()` method. Then call one of the per-level methods, such as `logging.debug()` or `logging.error()`, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file.

The file will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created.

The logger module provides 5 levels of logging messages, from DEBUG to CRITICAL. When you set up a logger, you specify the minimum level of messages to be logged. If you set up the logger with the minimum level set to ERROR, then only messages at ERROR and CRITICAL levels will be logged. Setting the minimum level to DEBUG allows all messages to be logged.

Table 1. Logging Levels

Level	Value
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`UNSET`	0

Example

logging_simple.py

```
import logging

logging.basicConfig(
    filename='../LOGS/simple.log',
    level=logging.WARNING,
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

../LOGS/simple.log

```
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
```

...

Formatting log entries

- Add format=format to basicConfig() parameters
- Format is a string containing directives and (optionally) other text
- Use directives in the form of %(item)type
- Other text is left as-is

To format log entries, provide a format parameter to the `basicConfig()` method. This format will be a string contain special directives (i.e. Placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form `%(item)type`, where `item` is the data field, and `type` is the data type.

To format the date from the `%(asctime)s` directive, assign a value to the `datefmt` parameter using date components from the table that follows.

Example

logging_formatted.py

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
set the format for log entries
    datefmt="%x-%X",
    filename='../LOGS/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.error("this is an ERROR")
value = 38.7
logging.error("Invalid value %s", value)
logging.info("this is information")
logging.critical("this is critical")
```

../LOGS/formatted.log

```
INFO root 09/26/25-13:51:02 logging_formatted.py 10 this is information
WARNING root 09/26/25-13:51:02 logging_formatted.py 11 this is a warning
ERROR root 09/26/25-13:51:02 logging_formatted.py 12 this is an ERROR
ERROR root 09/26/25-13:51:02 logging_formatted.py 14 Invalid value 38.7
INFO root 09/26/25-13:51:02 logging_formatted.py 15 this is information
CRITICAL root 09/26/25-13:51:02 logging_formatted.py 16 this is critical
```

Table 2. Log entry formatting directives

Directive	Description
`%(name)s`	Name of the logger (logging channel)
`%(levelno)s`	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
`%(levelname)s`	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available)
`%(filename)s`	Filename portion of pathname
`%(module)s`	Module (name portion of filename)
`%(lineno)d`	Source line number where the logging call was issued (if available)
`%(funcName)s`	Function name
`%(created)f`	Time when the LogRecord was created (time.time() return value)
`%(asctime)s`	Textual time when the LogRecord was created
`%(msecs)d`	Millisecond portion of the creation time
`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
`%(thread)d`	Thread ID (if available)
`%(threadName)s`	Thread name (if available)
`%(process)d`	Process ID (if available)
`%(message)s`	The result of record.getMessage(), computed just as the record is emitted

Table 3. Date Format Directives

Directive	Meaning	Notes
`%a`	Locale's abbreviated weekday name	
`%A`	Locale's full weekday name	
`%b`	Locale's abbreviated month name	
`%B`	Locale's full month name	
`%c`	Locale's appropriate date and time representation	
`%d`	Day of the month as a decimal number [01,31]	
`%f`	Microsecond as a decimal number [0,999999], zero-padded on the left	(1)
`%H`	Hour (24-hour clock) as a decimal number [00,23]	
`%I`	Hour (12-hour clock) as a decimal number [01,12]	
`%j`	Day of the year as a decimal number [001,366]	
`%m`	Month as a decimal number [01,12]	
`%M`	Minute as a decimal number [00,59]	
`%p`	Locale's equivalent of either AM or PM.	(2)
`%S`	Second as a decimal number [00,61]	(3)
`%U`	Week number (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0	(4)
`%w`	Weekday as a decimal number [0(Sunday),6]	
`%W`	Week number (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0	(4)
`%x`	Locale's appropriate date representation	
`%X`	Locale's appropriate time representation	
`%y`	Year without century as a decimal number [00,99]	
`%Y`	Year with century as a decimal number	
`%z`	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive)	(5)
`%Z`	Time zone name (empty string if the object is naive)	
`%%`	A literal '%' character	

Logging exception information

- Use logging.exception()
- Adds exception info to message
- Only in **except** blocks

The `logging.exception()` function will add a traceback to the log in addition to the specified message. It should only be called in an `except` block.

This is different from putting the file name and line number in the log entry. That puts the file name and line number where the logging method was called, while `logging.exception()` specifies the line where the error occurred.

Example

logging_exception.py

```
import logging

logging.basicConfig( # configure logging
    filename='../LOGS/exception.log',
    level=logging.WARNING,  # minimum level
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') # add exception info to the log
```

../LOGS/exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
10, in <module>
    result = i/0
             ~^~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
10, in <module>
    result = i/0
             ~^~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
10, in <module>
    result = i/0
             ~^~
ZeroDivisionError: division by zero
```

For more information

- https://docs.python.org/3/howto/logging.html
- https://docs.python.org/3/howto/logging-cookbook.html
- https://pymotw.com/3/logging/index.html

Chapter 1 Exercises

Exercise 1-1 (c2f_loop_safe.py)

Rewrite c2f_loop.py to handle the error that occurs if the user enters non-numeric data. The script should print an error message and go back to the top of the loop if an error occurs.

Exercise 1-2 (c2f_batch_safe.py)

Rewrite `c2f_batch.py` to handle the ValueError that occurs if sys.argv[1] is not a valid number. Log the error to a file named `c2f_batch.log`.

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

def read_files(*file_paths: str) -> None:
    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: # argument values must be ints
    for word, count in kwargs.items(): # loop through named arguments
        print(word.upper() * count)

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

hints_functions.py

```
a = 'PYTHONPYTHONPYTHON'
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

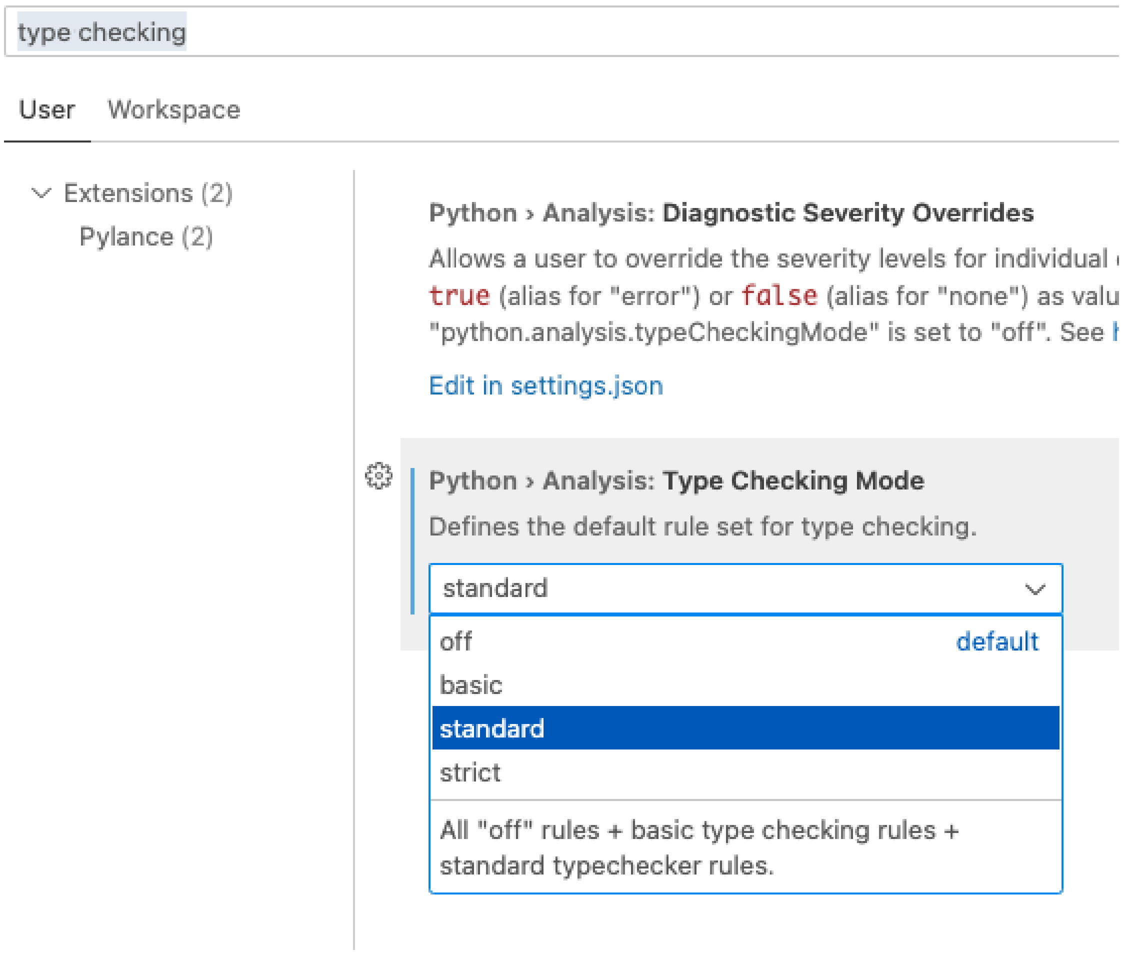
If these type hints are not used by the Python interpreter, how are they useful? While the Python interpreter does not (currently) use the type hints in any way, static analysis tools do. They check code *before* it is executed.

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.



type checking

Python › Analysis: Diagnostic Severity Overrides

Allows a user to override the severity levels for individual
`true` (alias for "error") or `false` (alias for "none") as valu
"python.analysis.typeCheckingMode" is set to "off". See

Python › Analysis: Type Checking Mode

Defines the default rule set for type checking.

standard

All "off" rules + basic type checking rules + standard typechecker rules.

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: Serializing Data

Objectives

- Read and write CSV data
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write YAML data
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes

Reading CSV data

- Use `csv` module
- Create `reader` object with file object or any iterable
- Iterate through reader to get rows as lists of columns

To read CSV data, first open the CSV file with the `open()` function to get a file object. Then create an instance of the `reader` class from the `csv` module, passing in the file object.

The `reader` object is an iterator over the rows (lines) in the file. It does not actually contain the data. Each row is returned as a list of the fields.

Any iterable with data in CSV format can be used in place of a file object.

Example

csv_read.py

```
import csv

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in)  # create CSV reader
    for name, title, color, quest, comment, number in rdr:  # Read and unpack records one
at a time; each record is a list
        print(f'{title:4s} {name:9s} {quest}')
```

csv_read.py

```
King Arthur    The Grail
Sir  Galahad   The Grail
Sir  Robin     Not Sure
Sir  Bedevere  The Grail
Sir  Gawain    The Grail
Sir  Hector    The Grrrrrail
Sir  Launcelot The Grail
```

...

Customizing CSV readers and writers

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to `csv.reader()` or `csv.writer()`. You can change the field and row delimiters, the escape character, and for output, what level of quoting. This can be used for any text file, not just CSV formats.

Dialects

You can also specify a *dialect*, which is a custom set of CSV parameters. TO create a custom dialect, use `csv.register_dialect()` with the name of the dialect, then specify `"dialect=dialect"` as a parameter to `csv.reader()` or `csv.writer()`.

You could put a dialect specification in a separate module, and import it as needed.

Table 4. CSV reader/writer Parameters

Parameter	**Meaning**
`quotechar`	One-character string to use as quoting character (default: `'"'`)
`delimiter`	One-character string to use as field separator (default: `','`)
`skipinitialspace`	If True, skip white space after field separator (default: `False`)
`lineterminator`	The character sequence which terminates rows (default: depends on OS)
`quoting`	When should quotes be generated when writing CSV `csv.QUOTE_MINIMAL` – only when needed (default) `csv.QUOTE_ALL` – quote all fields `csv.QUOTE_NONNUMERIC` – quote all fields that are not numbers `csv.QUOTE_NONE` – never put quotes around fields
`escapechar`	One-character string to escape delimiter when quoting is set to `csv.QUOTE_NONE`
`doublequote`	Control quote handling inside fields. When `True`, two consecutive quotes are read as one, and one quote is written as two. (default: `True`)
`dialect`	string representing registered dialect name, such as "excel"

Example

csv_nonstandard.py

```
import csv

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';')  # specify alternate field
delimiter

    # iterate over rows of data -- csv reader is an iterator

    for first_name, last_name, known_for, birth_date in rdr:
        print(f'{last_name}: {known_for}')
```

csv_nonstandard.py

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
van Rossum: Python
Kurtz: BASIC
Hopper: COBOL
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
van Rossum: Python
Lovelace:
Page: Google
Torvalds: Linux
```

Example

csv_dialects.py

```
import csv

csv.register_dialect('colon-sep', delimiter=":")

with open('../DATA/knights.txt') as knights_in:
    reader = csv.reader(knights_in, dialect="colon-sep")
    for row in reader:
        print(row)
print()

with open('../DATA/primeministers.txt') as pm_in:
    reader = csv.reader(pm_in, dialect="colon-sep")
    for row in reader:
        print(row)
```

csv_dialects.py

```
['Arthur', 'King', 'blue', 'The Grail', 'King of the Britons']
['Galahad', 'Sir', 'red', 'The Grail', "'I could handle some more peril'"]
['Lancelot', 'Sir', 'blue', 'The Grail', "It's too perilous!"]
['Robin', 'Sir', 'yellow', 'Not Sure', 'He boldly ran away']
['Bedevere', 'Sir', 'red, no blue!', 'The Grail', 'AARRRRRRRGGGGHH']
['Gawain', 'Sir', 'blue', 'The Grail', 'none']

['1', 'Sir John A.', 'Macdonald', '1867-7-1', '1873-11-5', 'Glasgow, Scotland', '1867-07-
01', '1873-11-05', 'Liberal-Conservative']
['2', 'Alexander', 'Mackenzie', '1873-11-7', '1878-10-8', 'Logierait, Scotland', '1873-
11-07', '1878-10-08', 'Liberal']
['3', 'Sir John A.', 'Macdonald', '1878-10-17', '1891-6-6', 'Glasgow, Scotland', '1878-
10-17', '1891-06-06', 'Liberal-Conservative']
```

Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the `DictReader` class.

By default, it will get field names from the first line of the file. Otherwise, you can specify a list of field names with the `fieldnames` parameter. For each row, you can then look up a field by name, rather than position.

Example

csv_dictreader.py

```
import csv

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] # field
names, which will become dictionary keys on each row

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names)  # create reader, passing
in field names (if not specified, uses first row as field names)
    for row in rdr:  # iterate over rows in file
        print(f"{row['firstname']:25} {row['lastname']:12} {row['party']}")
```

csv_dictreader.py

```
George                    Washington   no party
John                      Adams        Federalist
Thomas                    Jefferson    Democratic - Republican
James                     Madison      Democratic - Republican
James                     Monroe       Democratic - Republican
John Quincy               Adams        Democratic - Republican
Andrew                    Jackson      Democratic
Martin                    Van Buren    Democratic
William Henry             Harrison     Whig
John                      Tyler        Whig
James Knox                Polk         Democratic
Zachary                   Taylor       Whig
Millard                   Fillmore     Whig
Franklin                  Pierce       Democratic
James                     Buchanan     Democratic
Abraham                   Lincoln      Republican
Andrew                    Johnson      Republican
Ulysses Simpson           Grant        Republican
Rutherford Birchard       Hayes        Republican
James Abram               Garfield     Republican
```

...

Writing CSV Data

- Use `csv.writer()`
- Parameter is file-like object (must implement `write()` method)
- Can specify parameters to writer constructor
- Use `writerow()` or `writerows()` to output CSV data

To output data in CSV format, first open the output file for writing using `open()` with the `w` mode argument. Then create a writer by passing the file object to `csv.writer()`.

For each row to write, call the `writerow()` method of the writer, passing in an iterable with the values for that row. If you have your data in an iterable of iterables (a two-dimensional data structure) you can use `writerows()`.

On Windows, to prevent double-spaced output, add `lineterminator='\n'` when creating a CSV writer.

Any file-like object that implements the `write()` method can be used.

Example

csv_write.py

```
import sys
import csv

chicago_data = [
    ['Name', 'Position Title', 'Department', 'Employee Annual Salary'],
    ['BONADUCE,  MICHAEL J', 'POLICE OFFICER', 'POLICE', '$80724.00'],
    ['MELLON,  MATTHEW J "Matt"', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FIERI,  JOHN J', 'FIREFIGHTER-EMT', 'FIRE', '$75342.00'],
    ['GALAHAD,  MERLE S', 'CLERK III', 'BUSINESS AFFAIRS', '$45828.00'],
    ['ORCATTI,  JENNIFER L', 'FIRE COMMUNICATIONS OPERATOR I', 'OEMC', '$63121.68'],
    ['ASHE,  JOHN W', 'FOREMAN OF MACHINISTS', 'AVIATION', '$96553.60'],
    ['SADINSKY BLAKE,  MICHAEL G', 'POLICE OFFICER', 'POLICE', '$78012.00'],
    ['GRANT,  CRAIG A', 'SANITATION LABORER', 'STREETS & SAN', '$69576.00'],
    ['MILLER,  JONATHAN D', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FRANK,  ARTHUR R','POLICE OFFICER/EXPLSV DETECT, K9 HNDLR','POLICE','$87918.00'],
    ['POVOTTI,  JAMES S "Jimmy P"', 'TRAFFIC CONTROL AIDE-HOURLY', 'OEMC', '$19167.20'],
    ['TRAWLER,  DANIEL J', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SCUBA,  ANDREW G', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SWINE,  MATTHEW W', 'SERGEANT', 'POLICE', '$99756.00'],
    ['''RYDER,  MYRTA T "Lil'Myrt"''', 'POLICE OFFICER', 'POLICE', '$83706.00'],
    ['KORSHAK,  ROMAN', 'PARAMEDIC', 'FIRE', '$75372.00']
]

with open('../TEMP/chi_data.csv', 'w') as chi_out:
    #  On Windows, output line terminator must be set to '\n'.
    #  While it's not needed on Linux/Mac, it doesn't cause any problems,
    #  so this keeps the code portable.
    wtr = csv.writer(chi_out, lineterminator='\n') # create CSV writer from file object
that is opened
    for data_row in chicago_data:  # iterate over records from file
        data_row[0] = data_row[0].title()  # make first field title case rather than all
uppercase
        data_row[-1] = data_row[-1].lstrip('$')  # strip leading $ from last field
        wtr.writerow(data_row) # write one row (of iterables) to output file
```

Example

csv_writerows.py

```
import sys
import csv

chicago_data = [
    ['Name', 'Position Title', 'Department', 'Employee Annual Salary'],
    ['BONADUCE,  MICHAEL J', 'POLICE OFFICER', 'POLICE', '$80724.00'],
    ['MELLON,  MATTHEW J "Matt"', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FIERI,  JOHN J', 'FIREFIGHTER-EMT', 'FIRE', '$75342.00'],
    ['GALAHAD,  MERLE S', 'CLERK III', 'BUSINESS AFFAIRS', '$45828.00'],
    ['ORCATTI,  JENNIFER L', 'FIRE COMMUNICATIONS OPERATOR I', 'OEMC', '$63121.68'],
    ['ASHE,  JOHN W', 'FOREMAN OF MACHINISTS', 'AVIATION', '$96553.60'],
    ['SADINSKY BLAKE,  MICHAEL G', 'POLICE OFFICER', 'POLICE', '$78012.00'],
    ['GRANT,  CRAIG A', 'SANITATION LABORER', 'STREETS & SAN', '$69576.00'],
    ['MILLER,  JONATHAN D', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FRANK,  ARTHUR R','POLICE OFFICER/EXPLSV DETECT, K9 HNDLR','POLICE','$87918.00'],
    ['POVOTTI,  JAMES S "Jimmy P"', 'TRAFFIC CONTROL AIDE-HOURLY', 'OEMC', '$19167.20'],
    ['TRAWLER,  DANIEL J', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SCUBA,  ANDREW G', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SWINE,  MATTHEW W', 'SERGEANT', 'POLICE', '$99756.00'],
    ['''RYDER,  MYRTA T "Lil'Myrt"''', 'POLICE OFFICER', 'POLICE', '$83706.00'],
    ['KORSHAK,  ROMAN', 'PARAMEDIC', 'FIRE', '$75372.00']
]

with open('../TEMP/chi_data2.csv', 'w') as chi_out:
    wtr = csv.writer(chi_out, lineterminator='\n')
    wtr.writerows(chicago_data) # write all rows to output file
```

About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001 .

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

Reading JSON

- `json` module in standard library
- Parsing
  - `json.load()` from file object or file-like object
  - `json.loads()` parse from string
- Both methods return Python `dict` or `list`

To read a JSON file, import the `json` module. Use `json.loads()` to parse a string containing valid JSON. Use `json.load()` to read JSON from a file object (or a file-like object).

Both methods return a Python data structure (`dict` or `list`) containing all the data from the JSON file. The type of data structure returned depends on the outermost JSON data type in the file.

It is possible for a JSON document to be a single variable, but this is very uncommon. If this occurs, a single value will be returned, rather than a `dict` or `list`.

Example

json_read.py

```
from pprint import pprint
import json

# json.loads(STRING)     load from string
# json.load(FILE_OBJECT) load from file-like object

with open('../DATA/solar.json') as solar_in:  # open JSON file for reading
    solar = json.load(solar_in)  # load from file object and convert to Python data
structure

# uncomment to see raw Python data
# print('-' * 60)
# pprint(solar)
# print('-' * 60)
# print('\n\n')

print(solar['innerplanets'])  # solar is just a Python dictionary
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print("*" * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```

json_read.py

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name': 'Earth',
'moons': ['Moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
************************************************************
Mercury
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

Writing JSON

- Use json.dumps() or json.dump()

To create a JSON file from a Python data structure, open a text file for writing to get a file object. Then pass the data structure and the file object to `json.dump()`.

To create a JSON string, pass the data structure to `json.dumps()`.

Example

json_write.py

```
import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
]  # Python data structure

js = json.dumps(george, indent=4)  # dump structure to JSON string
print(js)

with open('george.json', 'w') as george_out:  # open file for writing
    json.dump(george, george_out, indent=4)  # dump structure to JSON file using open
file object
```

json_write.py

```
[
    {
        "num": 1,
        "lname": "Washington",
        "fname": "George",
        "dstart": [
            1789,
            4,
            30
        ],
        "dend": [
            1797,
            3,
            4
        ],
        "birthplace": "Westmoreland County",
        "birthstate": "Virginia",
        "dbirth": [
            1732,
            2,
            22
        ],
        "ddeath": [
            1799,
            12,
            14
        ],
        "assassinated": false,
        "party": null
    },
    {
        "spam": "ham",
        "eggs": [
            1.2,
            2.3,
            3.4
        ],
        "toast": {
            "a": 5,
            "m": 9,
            "c": 4
        }
    }
]
```

Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the `default` parameter ("default" is the actual name of the parameter) to `json.dump()` or `json.dumps()`.

The function should check the type of the object. If it is a type that needs special handling, return a JSON-friendly version, otherwise just return the original object.

Table 5. Python types that JSON can encode

Python	JSON
`dict`	object
`list` `tuple`	array
`str`	string
`int` `float`	number
`True`	true
`False`	false
`None`	null

see the file `json_custom_singledispatch.py` in EXAMPLES for how to use the `singledispatch` decorator (in the `functools` module) for another way to handle other data types.

Example

json_custom_encoding.py

```
import json
from datetime import date

class Parrot():  # sample user-defined class (not JSON-serializable)
    def __init__(self, name, color):
        self._name = name
        self._color = color

    @property
    def name(self):  # JSON does not understand arbitrary properties
        return self._name

    @property
    def color(self):
        return self._color

parrots = [  # list of Parrot objects
    Parrot('Polly', 'green'),  #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]

data = {  # dictionary of arbitrary data
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
}

# try to encode the Python data
try:
    print(json.dumps(data, indent=4))
except TypeError as err:
    print("ENCODING FAILED: ", err)
else:
    print("ENCODING SUCCESSFUL")
print('-' * 60)

def custom_encoder(obj):  # custom JSON encoder function
    if isinstance(obj, date):  # check for date object
        return obj.ctime()  # convert date to string
    elif isinstance(obj, Parrot):  # check for Parrot object
        return {'name': obj.name, 'color': obj.color}  # convert Parrot to dictionary
    return obj  # if not processed, return object for JSON to parse with default parser


# 'default' parameter specifies function for custom encoding;
try:
    print(json.dumps(data, default=custom_encoder, indent=4))
except TypeError as err:
    print("ENCODING FAILED: ", err)
else:
    print("ENCODING SUCCESSFUL")
```

json_custom_encoding.py

```
ENCODING FAILED:  Object of type date is not JSON serializable
------------------------------------------------------------
{
    "spam": [
        1,
        2,
        3
    ],
    "ham": [
        "a",
        "b",
        "c"
    ],
    "toast": "Fri Aug  1 00:00:00 2014",
    "parrots": [
        {
            "name": "Polly",
            "color": "green"
        },
        {
            "name": "Peggy",
            "color": "blue"
        },
        {
            "name": "Roger",
            "color": "red"
        }
    ]
}
ENCODING SUCCESSFUL
```

Reading and writing YAML

- yaml module from PYPI
- syntax like **json** module
- yaml.load(), dump() parse from/to file-like object
- yaml.loads(), dumps() parse from/to string

YAML is a structured data format which is a superset of JSON. However, YAML allows for a more compact and readable format.

Reading and writing YAML uses the same syntax as JSON, other than using the **yaml** module, which is NOT in the standard library. To install the **yaml** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(file_object)` or `yaml.loads(string)`.

To write a data structure to a YAML file or string, use `yaml.dump(data, file_object)` or `yaml.dumps(data)`.

You can also write custom YAML processors.

YAML parsers will parse JSON data

Example

yaml_load_solar.py

```
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in, Loader=yaml.Loader)

star = solar_data['star']
print(f"Our star is {star}\n")

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print(f"    {moon}")
```

yaml_load_solar.py

```
Our star is Sun

Mercury
    None
Venus
    None
Earth
    Moon
Mars
    Deimos
    Phobos
    Metis
Jupiter
    Adrastea
    Amalthea
    Thebe
    Io
    Europa
    Gannymede
    Callista
    Themisto
    Himalia
    Lysithea
    Elara
Saturn
    Rhea
    Hyperion
    Titan
    Iapetus
    Mimas
```

…

Example

yaml_dump.py

```
import sys
from datetime import date
import yaml

potus = {
    'presidents': [
         {
            'lastname': 'Washington',
            'firstname': 'George',
            'dob': date(1732, 2, 22),
            'dod': date(1799, 12, 14),
            'birthplace': 'Westmoreland County',
            'birthstate': 'Virginia',
            'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
            'assassinated': False,
            'party': None,
        },
        {
            'lastname': 'Adams',
            'firstname': 'John',
            'dob': date(1735, 10, 30),
            'dod': date(1826, 7, 4),
            'birthplace': 'Braintree, Norfolk',
            'birthstate': 'Massachusetts',
            'term': [date(1797, 3, 4), date(1801, 3, 4)],
            'assassinated': False,
            'party': 'Federalist',

        }
    ]
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```

yaml_dump.py

```
presidents:
- assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  firstname: George
  lastname: Washington
  party: null
  term:
  - 1789-04-30
  - 1797-03-04
- assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  firstname: John
  lastname: Adams
  party: Federalist
  term:
  - 1797-03-04
  - 1801-03-04
```

Which XML module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use xml.etree.ElementTree

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, use **lxml.etree**, which is based on **ElementTree** with some nice extra features, such as pretty-printing. While not part of the core Python library, it is provided by the Anaconda bundle.

If `lxml.etree` is not available, you can use **xml.etree.ElementTree** from the core library.

Getting Started With ElementTree

- Import xml.etree.ElementTree (or lxml.etree) as ET for convenience
- Parse XML or create empty ElementTree

ElementTree is part of the Python standard library; lxml is included with the Anaconda distribution.

Since putting "xml.etree.ElementTree" in front of its methods requires a lot of extra typing , it is typical to alias xml.etree.ElementTree to just ET when importing it: import xml.etree.ElementTree as ET

How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
  - Tag name
  - Attributes (implemented as a dictionary)
  - Text
  - Tail
  - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the get() method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

Only the tag property of an Element is required; other properties are optional.

Table 6. Element methods and properties

Method/Property	Description
`append(element)`	Add a subelement element to end of subelements
`attrib`	Dictionary of element's attributes
`clear()`	Remove all subelements
`find(path)`	Find first subelement matching path
`findall(path)`	Find all subelements matching path
`findtext(path)`	Shortcut for `find(path).text`
`get(attr)`	Get an attribute; Shortcut for `attrib.get()`
`getiterator()`	Returns an iterator over all descendants
`getiterator(path)`	Returns an iterator over all descendants matching path
`insert(pos,element)`	Insert subelement element at position pos
`items()`	Get all attribute values; Shortcut for `attrib.items()`
`keys()`	Get all attribute names; Shortcut for `attrib.keys()`
`remove(element)`	Remove subelement element
`set(attrib,value)`	Set an attribute value; shortcut for `attr[attrib] = value`
`tag`	The element's tag
`tail`	Text following the element
`text`	Text contained within the element

Table 7. ElementTree methods and properties

Property	Description
`find(path)`	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
`findall(path)`	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
`findtext(path)`	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
`getiterator(path)`	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
`getroot()`	Return the root node of the document
`parse(filename)` `parse(fileobj)`	Parse an XML source (filename or file-like object)
`write(filename,encoding)`	Writes XML document to filename, using encoding (Default us-ascii).

Parsing An XML Document

- Use ElementTree.parse()
- returns an ElementTree object
- Use get* or find* methods to select an element

Use the parse() method to parse an existing XML document. It returns an ElementTree object, from which you can find the root, or any other element within the document.

To get the root element, use the getroot() method.

Example

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```

Navigating the XML Document

- Use find() or findall()
- Element is iterable of it children
- findtext() retrieves text from element

To find the first child element with a given tag, use find('tag'). This will return the first matching element. The findtext('tag') method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the findall('tag') method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not None.

The ElementTree object also supports the find() and findall() methods of the Element object, searching from the root object.

Example

xml_planets_nav.py

```
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET


def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml')

    solar_system = doc.getroot()

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets')
    print('Inner:')

    for planet in inner:
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))


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

xml_planets_nav.py

```
<Element solarsystem at 0x1069a3a80>

Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
Dwarf:
     Pluto
```

Example

xml_read_movies.py

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml')  # read and parse the XML file

movies = movies_doc.getroot()  # get the root element (<movies>)

for movie in movies:  # loop through children of root element
    movie_name = movie.get('name'),  # get 'name' attribute of movie element
    movie_director = movie.findtext('director'),  # get 'director' attribute of movie
element
    print(f'{movie_name} by {movie_director}')
```

xml_read_movies.py

```
('Jaws',) by ('Spielberg, Stephen',)
('Vertigo',) by ('Alfred Hitchcock',)
('Blazing Saddles',) by ('Brooks, Mel',)
('Princess Bride',) by ('Reiner, Rob',)
('Avatar',) by ('Cameron, James',)
```

Using XPath

- Use simple XPath patterns Works with find* methods

When a simple tag is specified, the find* methods only search for subelements of the current element. For more flexible searching, the find* methods work with simplified **XPath** patterns. To find all tags named 'spam', for instance, use `.//spam`.

```
.//movie
presidents/president/name/last
```

Example

xml_planets_xpath1.py

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')  # parse XML file

inner_nodes = doc.findall('innerplanets/planet')  # find all elements (relative to root
element) with tag "planet" under "innerplanets" element

outer_nodes = doc.findall('outerplanets/planet')  # find all elements with tag "planet"
under "outerplanets" element

print('Inner:')
for planet in inner_nodes:  # loop through search results
    print('\t', planet.get("planetname"))  # print "name" attribute of planet element

print('Outer:')
for planet in outer_nodes:  # loop through search results
    print('\t', planet.get("planetname"))  # print "name" attribute of planet element
```

xml_planets_xpath1.py

```
Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
```

Example

xml_planets_xpath2.py

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')

jupiter = doc.find('.//planet[@planetname="Jupiter"]')

if jupiter is not None:
    for moon in jupiter:
        print(moon.text)  # grab attribute
```

xml_planets_xpath2.py

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Gannymede
Callista
Themisto
Himalia
Lysithea
Elara
```

Table 8. ElementTree XPath Summary

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
`*`	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
`.`	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
`//`	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “.//egg” selects all “egg” elements in the entire tree.
`..`	Selects the parent element.
`[@attrib]`	Selects all elements that have the given attribute. For example, “.//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
`[@attrib=▯value▯]`	Selects all elements for which the given attribute has the given value. For example, “.//div[@class=’sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
`parent_tag[child_tag]`	Selects all parent elements that has a child element named *child_tag*. In the current version, only a single tag can be used (i.e. only immediate children are supported). Parent tag can be *****.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression “last()” (for the last position), or a position relative to last() (e.g. “last()-1” for the second to last position). This predicate must be preceded by a tag name.

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml_create_knights.py** in the EXAMPLES folder

Example

xml_create_movies.py

```
# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization. This format is unique to Python, and a binary format.

The extension for pickle files is ".pkl", or sometimes ".pickle".

To create pickled data, use either pickle.dump() or pickle.dumps(). Both functions take a data structure as the first argument. dumps() returns the pickled data as a string. dump () writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use pickle.load(), which takes a file-like object that has been open for writing, or pickle.loads() which reads from a string. Both functions return the original data structure that had been pickled.

The syntax of the **json** module is based on the **pickle** module.

Example

pickling.py

```
import pickle
from pprint import pprint

# some data structures
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

values = [
    3/7, 1/9, 14.5
]

data = [  # list of data structures
    colors,
    airports,
    values,
]

print("BEFORE:")
pprint(data)
print('-' * 60)



with open('../TEMP/pickled_data.pkl', 'wb') as pkl_out:  # open pickle file for writing
in binary mode
    pickle.dump(data, pkl_out)  # serialize data structures to pickle file

with open('../TEMP/pickled_data.pkl', 'rb') as pkl_in:  # open pickle file for reading in
binary mode
    pickled_data = pickle.load(pkl_in)  # de-serialize pickle file back into data
structures

print("AFTER:")
pprint(pickled_data)  # view data structures
```

pickling.py

```
BEFORE:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
------------------------------------------------------------
AFTER:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
```

Chapter 3 Exercises

Exercise 3-1 (cpresidents.py)

Parse `presidents.csv`. Loop through and print out each president's first and last names and their political party.

Parse `presidents.json`. Loop through and print out each president's first and last names and their political party. === **Exercise 3-1 (jpresidents.py)**

Exercise 3-2 (ypresidents.py)

Parse `presidents.yaml`. Loop through and print out each president's first and last names and their political party.

Exercise 3-3 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
   <word>xanthan</word>
   <word>xanthans</words>
   and so forth
</words>
```

Exercise 3-4 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

Exercise 3-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into an dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the `pickle` module, Write the entire dictionary out to a file named `presidents.pkl``.

Exercise 3-6 (unpickle_potus.py)

Write a script to open presidents.pkl, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Chapter 4: Unit Testing with pytest

Objectives

- Understand the purpose of unit tests
- Design and implement unit tests with pytest
- Run tests in different ways
- Use builtin fixtures
- Create and use custom fixtures
- Mark tests for running in groups
- Learn how to mock data for tests

What is a unit test?

- Tests *unit* of code in isolation
- Ensures repeatable results
- Asserts expected behavior

A *unit test* is a test which asserts that an isolated piece of code (one function, method, class, or module) has some expected behavior. It is a way of making sure that code provides repeatable results.

There are four main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met
2. Test cases – collections of related unit tests
3. Fixtures — provide data to set up tests in order to get repeatable results
4. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Each test asserts that some condition is true.

Unit tests may collected into a **test case**, which is a related group of unit tests. With `pytest`, a test case can be either a module or a class.

Fixtures provide repeatable, known input to a test.

The final component is a **Test runner**, which executes one, some, or all tests and reports on the results. There are many different test runners for pytest. The builtin runner is very flexible.

The pytest module

- Provides
  - test runner
  - fixtures
  - special assertions
  - extra tools
- Not based on xUnit[1]

The `pytest` module provides tools for creating, running, and managing unit tests.

Here's how `pytest` implements the main components of unit testing:

unit test

A normal Python function that uses the `assert` statement to assert some condition is true

test case

A class that contains a group of related unit tests (tests can be also be grouped with *markers*).

fixture

A function that provides test resources (fixtures can be nested).

test runner

A text-based test runner is built in, and there are many third-party test runners

`pytest` is more flexible than classic **xUnit** implementations. For example, fixtures can be associated with any number of individual tests, or with a test class. Test cases need not be classes. These features help make it popular.

[1] The builtin unit testing module, `unittest`, *is* based on **xUnit** patterns, as implemented in Java and other languages. The `pytest` test runner will detect `unittest` tests as well. This can be handy for transitioning legacy code to `pytest`.

Creating tests

- Create test functions
- Use builtin `assert`
- Confirm something is true
- Optional message

To create a test, create a function whose name begins with "test". These should normally be in a separate script, whose name begins with "test_" or ends with "_test". For the simplest cases, tests do not even need to import `pytest`.

Each test function will use the builtin `assert` statement one or more times to confirm that that some condition is true. If all of the assertions are True, then the test passes. If any assertion fails, the test fails.

`pytest` will print an appropriate message by introspecting the expression, or you can add your own failure message after the expression, separated by a comma.

```
assert result == 100
assert 2 == 2, "Two is not equal to two!"
```

It is a good idea to make test names verbose. This will help when running tests in verbose mode, so you can see what tests are passing (or failing).

Real-life unit tests

`requests` is one of the most commonly used Python modules outside of the standard library. It provides an HTTP client with many helpful options. Here are some of the unit tests for `requests`:

https://github.com/psf/requests/blob/main/tests/test_requests.py

Example

tests/test_simple.py

```
# tests should begin with "test" for automatic discovery
def test_two_plus_two_equals_four():
    assert 2 + 2 == 4   # if assert statement succeeds, the test passes

def test_three_is_greater_than_two():
    assert 3 > 2
```

pytest -v tests/test_simple.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 2 items

tests/test_simple.py::test_two_plus_two_equals_four PASSED              [ 50%]
tests/test_simple.py::test_three_is_greater_than_two PASSED             [100%]

============================== 2 passed in 0.13s ===============================
```

Running tests (basics)

- Needs a test runner
- `pytest` provides `pytest` script

To actually run tests, you need a *test runner*. A test runner is software that runs one or more tests and reports the results.

`pytest` provides a script (also named `pytest`) to run tests from the command line (if using Anaconda, do this from an Anaconda prompt).

You can run a single test, a test case, a module, or all tests in a folder and all its subfolders.

```
pytest test_…py
```

to run the tests in a particular module, and

```
pytest -v test_…py
```

to add verbose output.

By default, pytest captures (and does not display) anything written to stdout/stderr. If you want to see the output of `print()` statements in your tests, add the **-s** option, which turns off output capture.

```
pytest -s …
```

If pytest does not run from the command line, use `python -m pytest`.

In older versions of pytest, the test runner script was named `py.test`. While newer versions support that name, the developers recommend only using `pytest`.

Visual Studio Code has a built-in test runner. It will normally automatically detect a `tests` folder in your project.

Special assertions

- Special cases
  - pytest.raises()
  - pytest.approx()

There are two special cases not easily handled by `assert`.

pytest.raises

For testing whether an exception is raised, use `pytest.raises()`. This should be used with the **with** statement:

```
with pytest.raises(ValueError):
    w = Wombat('blah')
```

The assertion will succeed if the code inside the **with** block raises the specified error.

pytest.approx

For testing whether two floating point numbers are *close enough* to each other, use `pytest.approx()`:

```
assert result == pytest.approx(1.55)
```

The default tolerance is 1e-6 (one part in a million). You can specify the relative or absolute tolerance to any degree. Infinity and NaN are special cases. NaN is normally not equal to anything, even itself, but you can specify `nanok=True` as an argument to approx().

See https://docs.pytest.org/en/latest/reference.html#pytest-approx for more information on pytest.approx()

Example

tests/test_special_assertions.py

```
import pytest
import math

FILE_NAME = 'IDONOTEXIST.txt'

#########################################################
# subject under test                                    #
#########################################################
def read_file_data(file_name):
    with open(file_name) as file_in:
        data = file_in.read().splitlines()
        return data
#########################################################


def test_missing_filename():
    """
    Assert FileNotFoundError is raised
    """
    with pytest.raises(FileNotFoundError):
        read_file_data(FILE_NAME)  # will pass test if file is NOT found


def test_floats_approximately_match():
    # fail unless values are within 0.000001 of each other
    # (actual result is 0.30000000000000004)
    assert (.1 + .2) == pytest.approx(.3)


def test_22_div_7_is_approximately_pi():
    # Default tolerance is 0.000001
    # smaller (or larger) tolerance can be specified
    assert 22 / 7 == pytest.approx(math.pi, .001)
```

pytest -v tests/test_special_assertions.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 3 items

tests/test_special_assertions.py::test_missing_filename PASSED           [ 33%]
tests/test_special_assertions.py::test_floats_approximately_match PASSED [ 66%]
tests/test_special_assertions.py::test_22_div_7_is_approximately_pi PASSED [100%]

============================== 3 passed in 0.13s ===============================
```

Fixtures

- Provide resources for tests
- Implement as functions
- Scope
  - Per test
  - Per class
  - Per module
- Source of fixtures
  - Builtin
  - User-defined

When writing tests for a particular object, many tests might require an instance of the object. This instance might be created with a particular set of arguments.

What happens if twenty different tests instantiate a particular object, and the object's API changes? Now you have to make changes in twenty different places.

To avoid duplicating code across many tests, pytest supports *fixtures*, which are functions that provide information to tests. The same fixture can be used by many tests, which lets you keep the fixture creation in a single place.

A fixture provides items needed by a test, such as data, functions, or class instances. A fixtures can be either builtin or custom.

What fixtures provide

Consistency
: test uses the same, repeatable data

Readability
: keeps test itself short and simple

Auto-use
: Reduces number of imports

Teardown
: Provides cleanup capabilities

Use `pytest --fixtures` to list all available builtin and user-defined fixtures.

User-defined fixtures

- Decorate with **pytest.fixture**
- Return value to be used in test
- Fixtures may be nested

To create a fixture, decorate a function with `pytest.fixture`. Whatever the function returns is the value of the fixture.

To use the fixture, pass it to the test function as a parameter. The return value of the fixture will be available as a local variable in the test.

Fixtures can take other fixtures as parameters as well, so they can be nested to any level.

It is convenient to put fixtures into a separate module so they can be shared across multiple test scripts.

Add docstrings to your fixtures and the docstrings will be displayed via `pytest --fixtures`

Example

tests/test_simple_fixture.py

```
import os
import pytest

from .silly import Silly

@pytest.fixture
def silly_object():
    return Silly()  # fixture returns instance of Silly

def test_silly_triples_value(silly_object):  # pass fixture as test parameter
    assert silly_object.triple(5) == 15

def test_silly_normalizes_string(silly_object):
    assert silly_object.normalize("   Spam   ") == "spam"
```

pytest -v tests/test_simple_fixture.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 2 items

tests/test_simple_fixture.py::test_silly_triples_value PASSED            [ 50%]
tests/test_simple_fixture.py::test_silly_normalizes_string PASSED        [100%]

============================== 2 passed in 0.12s ===============================
```

Builtin fixtures

- Variety of common fixtures
- Provide
  - Temp files and dirs
  - Logging
  - STDOUT/STDERR capture
  - Monkeypatching tools

Pytest provides a large number of builtin fixtures for common testing requirements.

Using a builtin fixture is like using user-defined fixtures. Just specify the fixture name as a parameter to the test. No imports are needed for this.

See https://docs.pytest.org/en/latest/reference.html#fixtures for details on builtin fixtures.

Example

tests/test_builtin_fixtures.py

```
import pytest

COUNTER_KEY = 'test_cache/counter'

def test_cache(cache):  # cache persists values between test runs
    value = cache.get(COUNTER_KEY, 0)
    print("\nCounter before:", value)
    cache.set(COUNTER_KEY, value + 1)  # cache fixture is similar to dictionary, but with
.set() and .get() methods
    value = cache.get(COUNTER_KEY, 0)  # cache fixture is similar to dictionary, but with
.set() and .get() methods
    print("\nCounter after:", value)
    assert True   # Make test successful

def hello():
    print("Hello, pytesting world")

def test_capsys(capsys):
    hello()  # Call function that writes text to STDOUT
    out, err = capsys.readouterr()  # Get captured output
    assert out == "Hello, pytesting world\n"

@pytest.mark.parametrize("num", range(3))
def test_temp_dir(tmpdir, num):
    print("TEMP DIR:", tmpdir, "\n")  # tmpdir fixture provides unique temporary folder
name
```

pytest -v tests/test_builtin_fixtures.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 5 items

tests/test_builtin_fixtures.py::test_cache PASSED                        [ 20%]
tests/test_builtin_fixtures.py::test_capsys PASSED                       [ 40%]
tests/test_builtin_fixtures.py::test_temp_dir[0] PASSED                  [ 60%]
tests/test_builtin_fixtures.py::test_temp_dir[1] PASSED                  [ 80%]
tests/test_builtin_fixtures.py::test_temp_dir[2] PASSED                  [100%]
```

```
============================== 5 passed in 0.13s ===============================
```

Table 9. Pytest Builtin Fixtures

Fixture	Brief Description
`cache`	Return cache object to persist state between testing sessions.
`capsys`	Enable capturing of writes (text mode) to `sys.stdout` and `sys.stderr`
`capsysbinary`	Enable capturing of writes (binary mode) to `sys.stdout` and `sys.stderr`
`capfd`	Enable capturing of writes (text mode) to file descriptors 1 and 2
`capfdbinary`	Enable capturing of writes (binary mode) to file descriptors 1 and 2
`doctest_namespace`	Return `dict` that will be injected into namespace of doctests
`pytestconfig`	Session-scoped fixture that returns `_pytest.config.Config` object.
`record_property`	Add extra properties to the calling test.
`record_xml_attribute`	Add extra xml attributes to the tag for the calling test.
`caplog`	Access and control log capturing.
`monkeypatch`	Return `monkeypatch` fixture providing monkeypatching tools
`recwarn`	Return `WarningsRecorder` instance that records all warnings emitted by test functions.
`tmp_path`	Return `pathlib.Path` instance with unique temp directory
`tmp_path_factory`	Return a `_pytest.tmpdir.TempPathFactory` instance for the test session.
`tmpdir`	Return `py.path.local` instance unique to each test
`tmpdir_factory`	Return `TempdirFactory` instance for the test session.

Configuring fixtures

- Create **conftest.py**
- Automatically included
- Provides
  - Fixtures
  - Hooks
  - Plugins
- Directory scope

The `conftest.py` file can be used to contain user-defined fixtures, as well as hooks and plugins. Subfolders can have their own conftest.py, which will only apply to tests in that folder.

In a test folder, define one or more fixtures in conftest.py, and they will be available to all tests in that folder, as well as any subfolders.

Hooks

Hooks are predefined functions that will automatically be called at various points in testing. All hooks start with *pytest_*. A pytest.Function object, which contains the actual test function, is passed into the hook.

For instance, `pytest_runtest_setup()` will be called before each test.

A complete list of hooks can be found here: https://docs.pytest.org/en/latest/reference.html#hooks

Plugins

There are many pytest plugins to provide helpers for testing code that uses common libraries, such as **Django** or **redis**.

You can register plugins in conftest.py like so:

```
pytest_plugins = "plugin1", "plugin2",
```

This will load the plugins.

Example

tests/conftest.py

```
#!/usr/bin/env python
from pytest import fixture
from .silly import Silly

@fixture
def silly_object():
    return Silly()  # fixture returns instance of Silly

# predefined hook (all hooks start with 'pytest_')
def pytest_runtest_setup(item):  # item is test function
    if "test_config" in str(item):
        print(f"Hello from setup, {item}", end=" ")
```

Example

tests/test_config.py

```
def test_one(silly_object):   # unit test that uses fixture from conftest.py
    assert silly_object.triple(5) == 15
    assert silly_object.normalize("   Spam   ") == "spam"
```

pytest -v tests/test_config.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 1 item

tests/test_config.py::test_one PASSED                                   [100%]

============================= 1 passed in 0.12s ==============================
```

Parametrizing tests

- Run same test on multiple values
- Add parameters to fixture decorator
- Test run once for each parameter
- Use `pytest.mark.parametrize()`

Many tests require testing a method or function against many values. Rather than writing a loop in the test, you can automatically repeat the test for a set of inputs via **parametrizing**.

Apply the `@pytest.mark.parametrize` decorator to the test. The first argument is a string with the comma-separated names of the parameters; the second argument is the list of parameters. The test will be called once for each item in the parameter list. If a parameter list item is a tuple or other multi-value object, the items will be passed to the test based on the names in the first argument.

For more advanced needs, when you need some extra work to be done before the test, you can do indirect parametrizing, which uses a parametrized fixture. See `test_parametrize_indirect.py` for an example.

The authors of pytest deliberately spelled it "parametrizing", not "parameterizing".

Example

tests/test_parametrization.py

```
import pytest
from .silly import Silly

@pytest.fixture
def silly_object():
    return Silly()  # fixture returns instance of Silly

# List of values for testing containing input and expected result
test_data = [
    (5, 15), ('a', 'aaa'), (False, False), (True, 3), (0, 0), ([True], [True, True,
True])
]

# Parametrize the test with the test data; the first argument is a string mapping
# parameters to the test data
@pytest.mark.parametrize("input,expected", test_data)
def test_triple(silly_object, input, expected):
    assert silly_object.triple(input) == expected   # Test the function with the
parameters
```

pytest -v tests/test_parametrization.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 6 items

tests/test_parametrization.py::test_triple[5-15] PASSED                  [ 16%]
tests/test_parametrization.py::test_triple[a-aaa] PASSED                 [ 33%]
tests/test_parametrization.py::test_triple[False-False] PASSED           [ 50%]
tests/test_parametrization.py::test_triple[True-3] PASSED                [ 66%]
tests/test_parametrization.py::test_triple[0-0] PASSED                   [ 83%]
tests/test_parametrization.py::test_triple[input5-expected5] PASSED      [100%]

============================== 6 passed in 0.13s ===============================
```

Marking tests

- Create groups of tests ("test cases")
- Can create multiple groups
- Use `@pytest.mark.somemark`

You can mark tests with labels so that they can be run as a group. Use `@pytest.mark.marker`, where *marker* is the marker (label), which can be any alphanumeric string.

Then you can select tests which contain or match the marker.

```
pytest -m "alpha"
pytest -m "not alpha"
pytest -m "alpha or beta"
pytest -m "alpha and not beta"
```

Registering markers

You can register markers in the **[pytest]** section of `pytest.ini`, so they will be listed, with a description, with `pytest --markers`:

```
[pytest]
markers =
  internet: test requires internet connection
  slow: tests that take more time (omit with '-m "not slow")
```

Example

tests/test_mark.py

```
import pytest

@pytest.mark.alpha  # Mark with label alpha
def test_one():
    assert 1

@pytest.mark.beta
@pytest.mark.alpha  # Mark with label alpha
def test_two():
    assert 1

@pytest.mark.gamma
@pytest.mark.beta  # Mark with label beta
def test_three():
    assert 1
```

pytest -v tests/test_mark.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 3 items

tests/test_mark.py::test_one PASSED                                      [ 33%]
tests/test_mark.py::test_two PASSED                                      [ 66%]
tests/test_mark.py::test_three PASSED                                    [100%]

============================== 3 passed in 0.12s ===============================
```

Running tests (advanced)

- Run all tests
- Run by
  - file name
  - class
  - function
  - name match
  - group (mark)

`pytest` provides many ways to select which tests to run.

Running all tests

By default, pytest will find and run all tests in the current and any descendent directories.

Use `-s` to disable capturing, so anything written to STDOUT is displayed. Use `-v` for verbose output.

```
pytest
pytest -v
pytest -s
pytest -vs
```

Running by component

Use the node ID to select by component, such as file, class, method, or function name:

```
file::class
file::class::test
file::test
```

```
pytest test_president.py::test_dates
pytest test_president.py::test_dates::test_birth_date
```

Note that the *file* name is specified, not the *module* name.

Running by name match

Use **-k** to run all tests where the file name, test name, or marker includes a specified string.

`pytest -k date` *run all tests whose name includes 'date'*

Making test files executable

While you should normally use `pytest` or your IDE to run tests, you can also make a test script run the tests when you execute the script normally with `python`. To do this, put the following code at the bottom of the test file:

```
if __name__ == '__main__':
    pytest.main([__file__, '-v'])   # Start the test runner
```

`__file__` is the name of the current file. You can add the '-s' option as another element in the list of arguments to `pytest.main`. You can also omit the '-v' option if you don't want verbose output.

Most of the time you should just use the test runner.

Skipping and failing

- Conditionally skip tests
- Completely ignore tests
- Decorate with
  - @pytest.mark.xfail
  - @pytest.mark.skip

To skip tests conditionally (or unconditionally), use `@pytest.mark.skip()`. This is useful if some tests rely on components that haven't been developed yet, or for tests that are platform-specific.

To fail on purpose, use `@pytest.mark.xfail)`. This reports the test as "XPASS" or "xfail", but does not provide traceback. Tests marked with xfail will not fail the test suite. This is useful for writing tests for not-yet-implemented features, or for testing objects with known bugs that will be resolved later.

Example

tests/test_skip.py

```
import sys
import pytest

def test_one():  # Normal test
    assert 1

# Unconditionally skip this test
@pytest.mark.skip(reason="can not currently test")
def test_two():
    assert 1


# Skip this test if current platform is not Windows
@pytest.mark.skipif(
    sys.platform != 'win32',
    reason="only implemented on Windows"
)
def test_three():
    assert 1


@pytest.mark.xfail
def test_four():
    assert 1


@pytest.mark.xfail
def test_five():
    assert 0
```

pytest -v tests/test_skip.py

```
============================ test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 5 items

tests/test_skip.py::test_one PASSED                                      [ 20%]
tests/test_skip.py::test_two SKIPPED (can not currently test)            [ 40%]
tests/test_skip.py::test_three SKIPPED (only implemented on Windows)     [ 60%]
tests/test_skip.py::test_four XPASS                                      [ 80%]
tests/test_skip.py::test_five XFAIL                                      [100%]

============== 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.16s ==============
```

Mocking data

- Simulate behavior of actual objects
- Replace expensive dependencies (time/resources)
- Use **pytest-mock**

Some objects have dependencies which can make unit testing difficult. These dependencies may be expensive in terms of time or resources, or can be slow to access.

One solution is to create fake dependencies, called *stubs*. A stub is a function or class that acts like the actual dependency, but has hard-coded data and doesn’t necessarily implement all features of the dependency.

A problem with stubs is that you end up with a lot of single-purpose stubs, which can be hard to maintain.

The better solution is to use **mock** objects, which pretend to be the real object. A mock object behaves like the original object, but is restricted and controlled in its behavior.

For instance, a class may have a dependency on a database query. A mock object may accept the query, but always returns a hard-coded set of results.

A mock object can record the calls made to it, and assert that the calls were made with correct parameters.

A mock object can be preloaded with a return value, or a function that provides dynamic (or random) return values.

Unlike a stub, a mock object is more elaborate, with record/playback capability, assertions, and many other test-related features.

Mocking in pytest

- Use `pytest-mock` plugin
  - Act as a stub
  - Patch existing code
- Can also use unittest.mock.Mock

The standard library provides `unittest.mock`. The `pytest-mock` plugin for `pytest` provides a wrapper around `unittest.mock`.

Once `pytest-mock` is installed, it provides a fixture named `mocker`, from which you can create mock objects.

In either case, there are two primary ways of mocking dependencies. One is to provide a replacement class, function, or data object that mimics the real thing.

The second is to patch a module, which temporarily (just during the test) replaces a component with a mock version. The `mocker.patch()` function replaces a component with a mock object. Any calls to the component are now recorded, and return values may be specified.

Mock where the object is used

- Don't patch original object
- Patch via subject under test

When the subject under test imports a module, it has a name that corresponds to that module. If you patch the original module, it only patches the original module, not the object pointed to by the subject under test.

For instance, if you are testing module `spam`, and it imports module `eggs` as follows, you might need to test that `spam` calls `eggs.toast`.

spam.py

```
from eggs import toast

toast()
```

Rather than patching `eggs.toast`, you will need to patch `spam.toast`. Always patch where the object is *used*, not where it's defined.

Mocking the correct name can be tricky. See https://nedbatchelder.com/blog/201908/why_your_mock_doesnt_work.html for more information.

Installing the modules to test

Before running the tests which use mocking, `spamlib` and `hamlib` must be locally installed on the computer where you're taking the class; otherwise tests won't be able to import them. You can install them in editable mode like this:

From the EXAMPLES folder:

```
cd hamlib
pip install -e .

cd ../spamlib
pip install -e .
```

Example

tests/test_mock.py

```
import pytest
import spamlib
from spamlib.spam import Spam

@pytest.fixture
def ham_value():
    return 42

@pytest.fixture
def ham_result(ham_value):  # use ham_value fixture
    return ham_value * 10

def test_spam_calls_ham(mocker, ham_value, ham_result):
    # need to patch spamlib.spam.ham, not hamlib.ham
    mocker.patch("spamlib.spam.ham", return_value=ham_result)
    s = Spam(ham_value)  # Create instance of Spam, which calls ham()
    assert s.value == ham_result
    assert spamlib.spam.ham.calledoncewith(ham_value)
```

pytest -v tests/test_mock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 0 items / 1 error

==================================== ERRORS ====================================
________________________ ERROR collecting test_mock.py _________________________
ImportError while importing test module
'/Users/jstrick/curr/courses/python/common/examples/tests/test_mock.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/importlib/__init__.py:1
26: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_mock.py:3: in <module>
    from spamlib.spam import Spam
E   ModuleNotFoundError: No module named 'spamlib.spam'
=========================== short test summary info ============================
ERROR tests/test_mock.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.14s ===============================
```

Example

tests/test_mock_pymock.py

```
from spamlib import spam
PATTERN = 'bug'
STRING_TO_SEARCH = 'lightning bug'

def test_spam_search_calls_re_search(mocker):   # Unit test
    # Patch re.search (i.e., replace re.search with a Mock object that
    # records calls to it)
    mocker.patch('spamlib.spam.re.search')

    s = spam.SpamSearch(PATTERN, STRING_TO_SEARCH)  # Create instance of SpamSearch
    s.findit()   # Call the method under test

    # Check that method was called just once with the expected parameters
    spam.re.search.assert_called_once_with(PATTERN, STRING_TO_SEARCH)
```

pytest -v tests/test_mock_pymock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 0 items / 1 error

==================================== ERRORS ====================================
_____________________ ERROR collecting test_mock_pymock.py _____________________
ImportError while importing test module
'/Users/jstrick/curr/courses/python/common/examples/tests/test_mock_pymock.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/importlib/__init__.py:1
26: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_mock_pymock.py:1: in <module>
    from spamlib import spam
E   ImportError: cannot import name 'spam' from 'spamlib' (unknown location)
=========================== short test summary info ============================
ERROR tests/test_mock_pymock.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.17s ===============================
```

Example

tests/test_mock_play.py

```
import pytest
from unittest.mock import Mock

@pytest.fixture
def small_list():   # Create fixture that provides a small list
    return [1, 2, 3]



def test_m1_returns_correct_list(small_list):
    m1 = Mock(return_value=small_list)  # Create mock object that "returns" a small list
    mock_result = m1('a', 'b')  # Call mock object with arbitrary parameters
    assert mock_result == small_list  # Check the mocked result



m2 = Mock()  # Create generic mock object

m2.spam('a', 'b')  # Call fake methods on mock object
m2.ham('wombat')  # Call fake methods on mock object
m2.eggs(1, 2, 3)  # Call fake methods on mock object

print("mock calls:", m2.mock_calls)  # Mock object remembers all calls

m2.spam.assert_called_with('a', 'b')  # Assert that spam() was called with parameters 'a'
and 'b'
```

pytest -v tests/test_mock_play.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: cov-6.0.0, anyio-4.0.0, Faker-37.1.0, mock-3.12.0, django-4.5.2
collecting ... collected 1 item

tests/test_mock_play.py::test_m1_returns_correct_list PASSED             [100%]

============================= 1 passed in 0.29s ==============================
```

Pytest plugins

- Common plugins
  - `pytest-qt`
  - `pytest-django`

There are some plugins for `pytest` that that integrate various frameworks which would otherwise be difficult to test directly.

The **pytest-qt** plugin provides a `qtbot` fixture that can attach widgets and invoke events. This makes it simpler to test your custom widgets.

The **pytest-django** plugin allows you to run Django with `pytest`-style tests rather than the default **unittest** style.

See https://docs.pytest.org/en/latest/reference/plugin_list.html for a complete list of plugins. There are currently 880 plugins!

Chapter 4 Exercises

Exercise 4-1 (test_president.py)

Using `pytest`, Create some unit tests for the President class you created earlier.[1]

Suggestions for tests:

- Create President objects for all current term numbers (1-*n*)
- What happens when an out-of-range term number is given?
- President 1’s first name is "George"
- All presidential terms match the correct last name (use list of last names and **parametrize**)
- Confirm date fields return an object of type **datetime.date**

[1] If there was not an exercise where you created a President class, you can use **president.py** in the top-level folder of the student guide.

Chapter 5: Regular Expressions

Objectives

- Using RE patterns
- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding option flags to a pattern
- Specifying capture groups
- Replacing text with backrefs and callbacks

Regular expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by **Perl**
- Python adds some conveniences

> Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as Does this string match the pattern?'', or Is there a match for the pattern anywhere in this string?''. You can also use REs to modify a string or to split it apart in various ways.
>
> — Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of **Perl** that they substantially changed from the originals. **Perl** added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses **Perl**-style regular expressions (AKA PCREs) and adds a few extensions of its own.

Two good web sites for creating and deciphering regular expressions:

Regex 101: https://regex101.com/#python
Pythex: http://www.pythex.org/

Two sites for having fun (yes, really!) with regular expressions:

Regex Golf: https://alf.nu/RegexGolf
Regex Crosswords: https://regexcrossword.com/

RE syntax overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more *branches* separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of *atoms*. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a *quantifier* (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.

There is frequently only one branch.

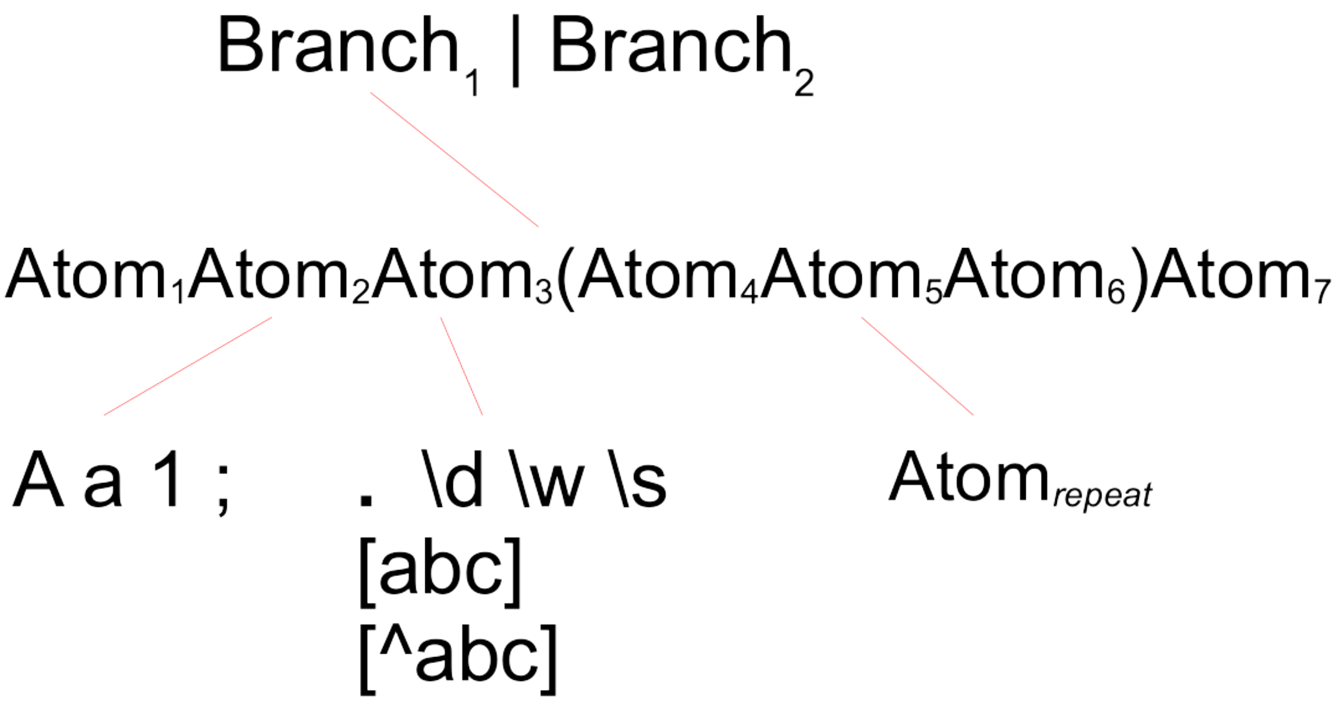


Table 10. Regular Expression Metacharacters

Pattern	Description
`.`	any character
`[abc]`	any character in set
`[^abc]`	any character not in set
`\w,\W`	any word, non-word char
`\d,\D`	any digit, non-digit
`\s,\S`	any space, non-space char
`^,$`	beginning, end of string
`\b`	beginning or end of word
`\`	escape a special character
`*,+,?`	0 or more, 1 or more, 0 or 1
`{m}`	exactly m occurrences
`{m,}`	at least m occurrences
`{m,n}`	m through n occurrences
`a|b`	match a or b
`(?aiLmsux)`	Set the A, I, L, M, S, U, or X flag for the RE (see below).
`(?:⋯)`	Non-capturing version of regular parentheses.
`(?P<name>⋯)`	The substring matched by the group is accessible by name.
`(?P=name)`	Matches the text matched earlier by the group named name.
`(?#⋯)`	A comment; ignored.
`(?=⋯)`	Matches if … matches next, but doesn't consume the string.
`(?!⋯)`	Matches if … doesn't match next.
`(?⇐⋯)`	Matches if preceded by … (must be fixed length).
`(?<!⋯)`	Matches if not preceded by … (must be fixed length).

Table 11. Regular Expression Examples

Pattern	Matches	Doesn't match	Comment
`pat`	`pat` `a lovely pattern` `compatible apps`	`pt` `poet` `pxt`	literal text
`p.t`	`pitter-patter` `keep to the right` `spiteful` `put your coat on`	`poet` `pt`	match any 1 char
`p..t`	`the` `poet` `a` `protein snack` `keep standing`	`pt` `potato`	matches any 2 chars
`p[aeiou]t`	`hotspot` `silly putty`	`pt` `poet`	any 1 vowel
`p[a-z]t`	`chopsticks` `spitball`	`stop,turn`	range `a` to `z`
`p[a-z0-9,]t`	`stop9truck` `hot spot` `stop,turn`	`p:t` `pt` `pAt`	two ranges + `,`
`p[aeiou][aeiou]t`	`poet` `repeated` `peatier`	`pot` `potato` `wipeout`	any vowel, any vowel
`[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]`	`236-17-1838` `492-12-7382`	`123-456-7890`	3 digits, 2 digits, 4 digits
`spam[^-]ham`	`spam ham` `spam#ham` `spamaham`	`spam-ham`	Any char not `-`
`\d\d\d-\d\d-\d\d\d\d`	`236-17-1838` `492-12-7382`	`123-456-7890`	3 digits, 2 digits, 4 digits
`pa+t`	`pat` `paaaat`	`pt`	one or more `a`
`pa*t`	`pt` `pat` `paaaat`	`pet` `pit`	zero or more `a`
`foo\s*bar`	`foobar` `foo bar` `foo      bar`	`foo-bar` `foo/bar`	zero or more spaces
`foo\s+bar`	`foo bar` `foo      bar`	`foobar`	one or more spaces
`foo\s?bar`	`foobar` `foo bar`	`foo   bar` `foobar`	zero or one space
`\d{3}-\d{2}-\d{4}`	`236-17-1838` `492-12-7382`	`123-456-7890`	3 digits, 2 digits, 4 digits
`p[aeiou]{1,2}t`	`compatible` `poetry`	`proton` `potato`	one or two vowels
`pa{2,}t`	`paat` `paaaat` `paaaaaat`	`pt` `pat` `pet`	at least two `a`s
`St\.`	`St. Francis`	`Ste Francis` `Stop`	\. matches only `.`
`^\d+$`	`123` `4`	`abcdef123xyz`	only digits
`^\s*\d+\s*$`	`123` `  123  ` `123  `	`12 3` `  123m`	\s* zero or more spaces
`^foo`	`football` `foobar`	`tenderfoot`	match beginning of string

Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

Find first match

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

Searches s and returns the first match. Returns a match object (**SRE_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

Find all matches

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

Provides a match object for each match found. Normally used with a **for** loop.

Retrieve text of all matches

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

Finds all matches and returns a list of matched strings. Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

Other search methods

`re.match()` is like `re.search()`, but searches for the pattern at beginning of `s`. There is an implied `^` at the beginning of the pattern.

Likewise `re.fullmatch()` only succeeds if the pattern matches the entire string. `^` and `$` around the pattern are implied.

Example

regex_finding_matches.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'  # store pattern in raw string

if re.search(pattern, s):  # search returns True on match
    print("Found pattern.")
print()

m = re.search(pattern, s)  # search actually returns match object
print(m)
if m:
    print("Found:", m.group(0))  # group(0) returns text that was matched by entire
expression (or just m.group())
print()

for m in re.finditer(pattern, s):  # iterate over all matches in string:
    print(m.group())
print()

matches = re.findall(pattern, s)  # return list of all matches
print("matches:", matches)
```

regex_finding_matches.py

```
Found pattern.

<re.Match object; span=(12, 17), match='M-302'>
Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

RE objects

- `re` object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An `re` object is created by calling `re.compile()` with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. `re.compile()` has an optional argument for flags which enable special features or fine-tune the match.

It is generally a good practice to create your re objects in a location near the top of your script, and then use them as necessary

Example

regex_objects.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]-\d{2,3}')  # Create an re (regular expression) object

if rx_code.search(s):  # Call search() method from the object
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

regex_objects.py

```
Found pattern.

Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

Compilation flags

- Fine-tune match
- Add readability

When using functions from `re`, or when compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined with `|`, the bitwise ***OR*** operator. Each flag has a short for and a long form.

Ignoring case

```
re.I, re.IGNORECASE
```

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the LOCALE flag.

Using the locale

```
re.L, re.LOCALE
```

Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write \w+ to match words, but \w only matches the character class `[A-Za-z]`; it won't match "é" or "ç". If your system is configured properly and a French locale is selected, certain C functions will tell the program that "é" should also be considered a letter. Setting the `LOCALE` flag enables \w+ to match French words as you'd expect.

Ignoring whitespace

```
re.X, re.VERBOSE
```

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a `#` that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

```
RE_ENTRY = r"""
    (?P<month>[A-Z][a-z]{2})\s+(?P<day>\d+)\s+              # date
    (?P<hour>\d{2}):(?P<minute>\d{2}):(?P<second>\d{2})\s+  # timestamp
    (?P<hostname>\S+?)\s+                                   # hostname
    (?P<process_name>.*?)                                   # process name
    \[(?P<pid>\d+)\]:\s+                                    # PID
    (?P<message>.*)                                         # message
"""
```

Example

regex_flags.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'

if re.search(pattern, s, re.IGNORECASE):  # make search case-insensitive
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I | re.M)  # short version of flag
if m:
    print("Found:", m.group())
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

regex_flags.py

```
Found pattern.

Found: M-302

M-302
r-99
H-476
Q-51
z-883
A-110
H-332
Y-45

matches: ['M-302', 'r-99', 'H-476', 'Q-51', 'z-883', 'A-110', 'H-332', 'Y-45']
```

Working with enbedded newlines

- `re.S`, `re.DOTALL` lets `.` match newline
- `re.M`, `re.MULTILINE` lets `^` and `$` match lines

Some text contains newlines (`\n`), representing mutiple lines within the string. There are two regular expression flags you can use with `re.search()` and other functions to control how they are searched.

These flags are not useful if the string has no embedded newlines.

Treating text like a single string

By default, `.` does not match newline. Thus, `spam.*ham` will not match the text if `spam` is on one line and `ham` is on a subsequent line. The `re.DOTALL` flag allows `.` to match newline, enabling searches that span lines.

`re.S` is an abbreviation for `re.DOTALL`.

Treating text like multiple lines

Normally, `^` only matches the beginning of a string and `$` only matches the end. If you use the `re.MULTILINE` flag, these anchors will also match the beginning and end of embedded lines.

`re.M` is an abbrevation for `re.MULTILINE`.

Example

regex_newlines.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

line_start_word = r'^\w+'  # match word at beginning of string/line

matches = re.findall(line_start_word, s)  # only matches at beginning of string
print("matches:", matches)
print()

matches = re.findall(line_start_word, s, re.M)  # matches at beginning of lines
print("matches:", matches)
print()

phrase = r"aliquip.*commodo"

match = re.search(phrase, s)
if match:
    print(match.group(), match.start())
else:
    print(f"{phrase} not found")
print()

match = re.search(phrase, s, re.S)
if match:
    print(repr(match.group()), match.start())
else:
    print(f"{phrase} not found")
print()
```

regex_newlines.py

```
matches: ['lorem']

matches: ['lorem', 'ad', 'ea', 'voluptate', 'Excepteur', 'officia']

aliquip.*commodo not found

'aliquip ex  \nea commodo' 223
```

Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with match.group()

Sometimes you need to grab just *part* of the text matched by an RE. Your pattern can contain one or more subgroups which match the parts you're interested in. For example, you could pull the hour, minute, and second separately out of the text "11:13:42". You can write a pattern that matches the entire time string and has a group for each part:

```
r"(\d{2}):(\d{2}):(\d{2})""
```

Groups are marked with parentheses, and "capture" whatever matched the pattern inside the parentheses.

`re.findall()` returns a list of tuples, where each tuple contains the matches for all each group.

To access groups in more detail, use `re.finditer()` and call the `.group()` method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either `match.group(0)`, or just `match.group()`. Then, `match.group(1)` returns text matched by the first set of parentheses, `match.group(2)` returns the text from the second set, etc.

In the same vein, `match.start()` or `match.start(0)` return the beginning 0-based offset of the entire match; `match.start(1)` returns the beginning offset of group 1, and so forth. The same is true for `match.end()` and `match.end(n)`.

`match.span()` returns the the start and end offsets for the entire match. `match.span(1)` returns start and end offsets for group 1, and so forth.

Example

regex_group.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'([A-Z])-(\d{2,3})'  # parens delimit groups

print("Group 0            Group 1              Group 2")
header2 = "text  start  end   text  start  end     text  start  end"
print(header2)
print("-" * len(header2))

for m in re.finditer(pattern, s):
    print(
        f"{m.group(0):5s}  {m.start(0):3d}  {m.end(0):3d}"
        f"    {m.group(1):5s}  {m.start(1):3d}  {m.end(1):3d}"
        f"      {m.group(2):5s}  {m.start(2):3d}  {m.end(2):3d}"
    )
print()

matches = re.findall(pattern, s)  # findall() returns list of tuples containing groups
print("matches:", matches)
```

regex_group.py

```
Group 0            Group 1              Group 2
text  start  end   text  start  end     text  start  end
--------------------------------------------------------
M-302   12   17    M       12   13      302     14   17
H-476  102  107    H      102  103      476    104  107
Q-51   134  138    Q      134  135      51     136  138
A-110  398  403    A      398  399      110    400  403
H-332  436  441    H      436  437      332    438  441
Y-45   470  474    Y      470  471      45     472  474

matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('A', '110'), ('H', '332'), ('Y',
'45')]
```

Special groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark.

Non-capture groups

The most basic special is `(?:pattern)`, which groups but does not capture.

Named groups

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax ``(?`P<name>pattern)``. You can then call `match.group("name")` to fetch the text match by that sub-expression; alternatively, you can call `match.groupdict()`, which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

Lookaround assertions

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax `(?=pattern)`. The string being matched must match the lookahead, but does not become part of the overall match.

For instance, `\d(?st|nd|rd|th)(?=street)` matches "1st", "2nd", etc., but only where they are followed by "street".

Example

regex_special.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])-(?P<number>\d{2,3})'  # Use (?P<NAME>...) to name groups

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number'))  # Use m.group(NAME) to retrieve text
```

regex_special.py

```
M 302
H 476
Q 51
A 110
H 332
Y 45
```

Replacing text

- Use `re.sub(pattern, replacement,string[,count])`
- `re.subn(…)` returns string and count

To find and replace text using a regular expression, use the `re.sub()` method. It takes the pattern, the replacement text and the string to search as arguments, and returns the modified string.

The third (optional) argument is one or more compilation flags. The fourth argument is the maximum number of replacements to make. Both are optional, but if the fourth argument is specified and no flags are needed, use `0` as a placeholder for the third argument.

Be sure to put the arguments in the proper order!

Example

regex_sub.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) # replace pattern with string
print(s2)
print()

s3, count = rx_code.subn("___", s) # subn returns tuple with result string and
replacement count
print(f"Made {count} replacements")
print()
print(s3)
```

regex_sub.py

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed do
 eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua. Ut
enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [REDACTED]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa qui
officia deserunt [REDACTED] mollit anim id est laborum

Made 8 replacements

lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do
 eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo ___  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
officia deserunt ___ mollit anim id est laborum
```

Replacing with backrefs

- Use match or groups in replacement text
  - `\g<num>` *group number*
  - `\g<name>` *group name*
  - `\num` *shortcut for group number*

It is common to need all or part of the match in the replacement text. To allow this, the `re` module provides *backrefs*, which are special variables that *refer back* to the groups in the match, including group 0 (the entire match).

To refer to a particular group, use the syntax `\g<num>`.

To refer to a particular named group, use `\g<name>`.

As a shortcut, you can use `\num`. This does not work for group 0 — use `\g<0>` instead.

`\10` is group 10, not group 1 followed by '0'

Example

regex_sub_backrefs.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub(r"(\g<1>)[\g<2>]", s)
print(f"s2: {s2}")
print('-' * 60)

s3 = rx_code.sub(r"\g<number>-\g<letter>", s)
print(f"s3: {s3}")
print('-' * 60)

s4 = rx_code.sub(r"[\1:\2]", s)
print(f"s4: {s4}")
print('-' * 60)
```

regex_sub_backrefs.py

```
s2: lorem ipsum (M)[302] dolor sit amet, consectetur (r)[99] adipiscing elit, sed do
 eiusmod tempor incididunt (H)[476] ut labore et dolore magna (Q)[51] aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo (z)[883]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat (A)[110] cupidatat non proident, sunt in (H)[332] culpa qui
officia deserunt (Y)[45] mollit anim id est laborum
------------------------------------------------------------
s3: lorem ipsum 302-M dolor sit amet, consectetur 99-r adipiscing elit, sed do
 eiusmod tempor incididunt 476-H ut labore et dolore magna 51-Q aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo 883-z  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat 110-A cupidatat non proident, sunt in 332-H culpa qui
officia deserunt 45-Y mollit anim id est laborum
------------------------------------------------------------
s4: lorem ipsum [M:302] dolor sit amet, consectetur [r:99] adipiscing elit, sed do
 eiusmod tempor incididunt [H:476] ut labore et dolore magna [Q:51] aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [z:883]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat [A:110] cupidatat non proident, sunt in [H:332] culpa qui
officia deserunt [Y:45] mollit anim id est laborum
------------------------------------------------------------
```

Replacing with a callback

- Replacement can be a callback function
- Function expects match object, returns replacement text
- Use either normally defined function or a lambda

In addition to using a string, possibly containing backrefs, as the replacement, you can specify a function. This function will be called once for each match, with the match object as its only parameter.

Using a callback is necessary if you need to modify any of the original text.

Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to:

- add text around the replacement (can also be done with backrefs)
- search ignoring case and preserve case in a replacement
- look up the text in a dictionary or database to find replacement text

Example

regex_sub_callback.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

def update_code(match):  # callback function is passed each match object
    letter = match.group('letter').upper()
    number = int(match.group('number'))
    return f'{letter}:{number:04d}'  # function returns replacement text



s2, count = rx_code.subn(update_code, s)  # sub takes callback function instead of
replacement text
print(s2)
print(count, "replacements made")
```

regex_sub_callback.py

```
lorem ipsum M:0302 dolor sit amet, consectetur R:0099 adipiscing elit, sed do
 eiusmod tempor incididunt H:0476 ut labore et dolore magna Q:0051 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z:0883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A:0110 cupidatat non proident, sunt in H:0332 culpa qui
officia deserunt Y:0045 mollit anim id est dlaborum
8 replacements made
```

Splitting a string

- Syntax: `re.split(pattern, string[,max])`

The `re.split()` method splits a string into pieces, using the regex to match the delimiters, and returning the pieces as a list. The optional `max` argument limits the numbers of pieces.

Example

regex_split.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""


# pattern is one or more non-letters
rx_wordsep = re.compile(r"[^a-z0-9-]+", re.I)  # When splitting, pattern matches what you
don't want

words = rx_wordsep.split(s)  # Retrieve text _separated_ by your pattern
unique_words = set(words)

print(sorted(unique_words))
```

regex_split.py

```
['A-110', 'Duis', 'Excepteur', 'H-332', 'H-476', 'M-302', 'Q-51', 'U901', 'Ut', 'Y-45',
'ad', 'adipiscing', 'aliqua', 'aliquip', 'amet', 'anim', 'aute', 'cillum', 'commodo',
'consectetur', 'consequat', 'culpa', 'cupidatat', 'deserunt', 'dlaborum', 'do', 'dolor',
'dolore', 'ea', 'eiusmod', 'elit', 'enim', 'esse', 'est', 'et', 'eu', 'ex',
'exercitation', 'fugiat', 'id', 'in', 'incididunt', 'ipsum', 'irure', 'labore',
'laboris', 'lorem', 'magna', 'minim', 'mollit', 'nisi', 'non', 'nostrud', 'nulla',
'occaecat', 'officia', 'pariatur', 'proident', 'qui', 'quis', 'r-99', 'reprehenderit',
'sed', 'sint', 'sit', 'sunt', 'tempor', 'ullamco', 'ut', 'velit', 'veniam', 'voluptate',
'z-883']
```

Chapter 5 Exercises

Exercise 5-1 (pyfind.py)

Write a script which takes two or more arguments. The first argument is the pattern to search for; the remaining arguments are files to search. For each file, print out all lines which match the pattern. [1]

Example

```
python pyfind.py freezer DATA/words.txt DATA/parrot.txt
```

Exercise 5-2 (mark_big_words_callback.py, mark_big_words_backrefs.py)

Copy `parrot.txt` to `bigwords.txt` adding asterisks around all words that are 8 or more characters long.

HINT: Use the `\b` anchor to indicate beginning or end of a word.

There are two solutions to this exercise in the ANSWERS folder: one using a callback function, and one using backrefs.

Exercise 5-3 (print_numbers.py)

Write a script to print out all lines in `custinfo.dat` which contain phone numbers. A phone number consists of three digits, a dash, and four more digits.

Exercise 5-4 (word_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression `[^\w']+` for splitting each line into words.

Test with any of the text files in the DATA folder.

HINT: Use a dictionary for counting.

The specified pattern matches one or more characters that are neither letters, digits, underscores, nor apostrophes.

[1] Any similarity to the Unix `grep` command is purely intentional.

Chapter 6: Pythonic Programming

Objectives

- Learn what makes code "Pythonic"
- Understand some Python-specific idioms
- Create lambda functions
- Perform advanced slicing operations on sequences
- Distinguish between collections and generators

The Zen of Python

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than **right** now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea — let's do more of those!
>
> — Tim Peters, from PEP 20

Tim Peters is a longtime contributor to Python. He wrote the standard sorting routine, known as **timsort**.

The above text is printed out when you execute the code `import this`. Generally speaking, if code follows the guidelines in the Zen of Python, then it's Pythonic.

Tuples

- Fixed-size, read-only
- Collection of related items
- Supports some sequence operations
- Think 'struct' or 'record'

A **tuple** is a collection of related data. While on the surface it seems like just a read-only list, it is used when you need to pass multiple values to or from a function, but the values are not all the same type

To create a tuple, use a comma-separated list of objects. Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.

A tuple in Python might be represented by a struct or a "record" in other languages.

While both tuples and lists can be used for any data:

- Use a list when you have a collection of similar objects.
- Use a tuple when you have a collection of related objects, which may or may not be similar.

To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object: color = 'red',

Example

```
hostinfo = ( 'gemini','linux','ubuntu','hardy','Bob Smith' )

birthday = ( 'April',5,1978 )
```

Iterable unpacking

- Copy iterable to list of variables
- Frequently used with list of tuples
- Make code more readable

When you have an iterable such as a tuple or list, you access individual elements by index. However, `spam[0]` and `spam[1]` are not so readable compared to `first_name` and `company`. To copy an iterable to a list of variable names, just assign the iterable to a comma-separated list of names:

```
birthday = ( 'April',5,1978 )
month, day, year = birthday
```

You may be thinking "why not just assign to the variables in the first place?". For a single tuple or list, this would be true. The power of unpacking comes in the following areas:

- Looping over a sequence of tuples
- Passing tuples (or other iterables) into a function

Example

unpacking_people.py

```
people = [
    ('Melinda', 'Gates', 'Gates Foundation', '1964-08-15'),
    ('Steve', 'Jobs', 'Apple', '1955-02-24'),
    ('Larry', 'Wall', 'Perl', '1954-09-27'),
    ('Paul', 'Allen', 'Microsoft', '1953-01-21'),
    ('Larry', 'Ellison', 'Oracle', '1944-08-17'),
    ('Grace', 'Hopper', 'COBOL', '1906-12-09'),
    ('Bill', 'Gates', 'Microsoft', '1955-10-28'),
    ('Mark', 'Zuckerberg', 'Facebook', '1984-05-14'),
    ('Sergey','Brin', 'Google', '1973-08-21'),
    ('Larry', 'Page', 'Google', '1973-03-26'),
    ('Linus', 'Torvalds', 'Linux', '1969-12-28'),
]


# Unpack each tuple into four variables.
for first_name, last_name, product, dob in people:
    print(f"{first_name} {last_name}")
```

unpacking_people.py

```
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Grace Hopper
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

Extended iterable unpacking

- Allows for one "wild card"
- Allows common "first, rest" unpacking

When unpacking iterables, sometimes you want to grab parts of the iterable as a group. This is provided by extended iterable unpacking.

One (and only one) variable in the result of unpacking can have a star prepended. This variable will be a list of all values not assigned to other variables.

Example

extended_iterable_unpacking.py

```
values = ['a', 'b', 'c', 'd', 'e']  # values has 6 elements

x, y, *z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

x, *y, z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

*x, y, z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for *name, _ in people:  # name gets all but the last field
    print(name)
print()
```

extended_iterable_unpacking.py

```
x: a    y: b    z: ['c', 'd', 'e']

x: a    y: ['b', 'c', 'd']    z: e

x: ['a', 'b', 'c']    y: d    z: e

['Bill', 'Gates']
['Steve', 'Jobs']
['Paul', 'Allen']
['Larry', 'Ellison']
['Mark', 'Zuckerberg']
['Sergey', 'Brin']
['Larry', 'Page']
['Linux', 'Torvalds']
```

Unpacking function arguments

- Go from iterable to list of items
- Use * or **

Sometimes you need the other end of iterable unpacking. What do you do if you have a list of three values, and you want to pass them to a method that expects three positional arguments? One approach is to use the individual items by index. A more Pythonic approach is to use * to *unpack* the iterable into individual items:

Use a single asterisk to unpack a list or tuple (or similar iterable); use two asterisks to unpack a dictionary or similar.

In the example, see how the list **HEADINGS** is passed to .format(), which expects individual parameters, not *one parameter* containing multiple values.

Example

unpacking_function_args.py

```
people = [ # list of 4-element tuples
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Brown', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

def display_person(first_name, last_name, city, state): # function with four parameters
    print(f"{first_name:10} {last_name:10} {city:10} {state}")

display_person("Wanda", "Lefkowitz", "Albany", "NY")  # requires four arguments

for person in people:  # person is a tuple (one element of people list)
    display_person(*person)  # *person unpacks the tuple into four individual arguments
print()

def add_user(*, first_name, last_name, user_id):
    print(f"adding {first_name} {last_name} {user_id}")

user_info = {
    "user_id": "potus-16",
    "last_name": "Lincoln",
    "first_name": "Abraham",
}

add_user(**user_info)
```

unpacking_function_args.py

```
Wanda      Lefkowitz  Albany     NY
Joe        Schmoe     Burbank    CA
Mary       Brown      Madison    WI
Jose       Ramirez    Ames       IA

adding Abraham Lincoln potus-16
```

The sorted() function

- Returns a sorted copy of any collection
- Customize with named keyword parameters

```
key=
reverse=
```

The sorted() builtin function returns a sorted copy of its argument, which can be any iterable.

You can customize sorted with the **key** parameter.

Example

basic_sorting.py

```
"""Basic sorting example"""

fruits = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
          "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear", "banana",
          "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry", "lychee",
          "grape"]

sorted_fruit = sorted(fruits)  # sorted() returns a list

print(f"{sorted_fruit = }\n")

nums = [800, 80, 1000, 32, -3, 8, 18, 255, 400, 5, 5000]

print(f"{sorted(nums) = }")
```

basic_sorting.py

```
sorted_fruit = ['Apple', 'BLUEberry', 'FIG', 'Kiwi', 'ORANGE', 'Tamarind', 'Watermelon',
'apricot', 'banana', 'cherry', 'date', 'elderberry', 'grape', 'guava', 'lemon', 'lime',
'lychee', 'papaya', 'peach', 'pear', 'persimmon', 'pomegranate']

sorted(nums) = [-3, 5, 8, 18, 32, 80, 255, 400, 800, 1000, 5000]
```

Custom sort keys

- Use `key` parameter
- Specify name of function to use
- Key function takes exactly one parameter
- Useful for case-insensitive sorting, sorting by external data, etc.

You can specify a function with the `key` parameter of the `sorted()` function. This function will be used once for each element of the list being sorted, to provide the comparison value. Thus, you can sort a list of strings case-insensitively, or sort a list of zip codes by the number of Starbucks within the zip code.

The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value or a tuple of values. The returned values will be compared in order.

You can use any builtin Python function or method that meets these requirements, or you can write your own function.

The `lower()` method can be called directly from the builtin object `str`. It takes one string argument and returns a lower case copy.

```
sorted_strings = sorted(unsorted_strings, key=str.lower)
```

Example

custom_sort_keys.py

```
fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon",
         "Kiwi", "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG",
         "pear", "banana", "Tamarind", "persimmon", "elderberry", "peach",
         "BLUEberry", "lychee", "grape"]

def ignore_case(item):  # Parameter is _one_ element of iterable to be sorted
    return item.lower()  # Return value to sort on

fs1 = sorted(fruit, key=ignore_case)  # Specify function with named parameter key
print("Ignoring case:")
print(f"fs1: {fs1}\n")



def by_length_then_name(item):
    return (len(item), item.lower())  # Key functions can return tuple of values to
compare, in order

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(f"fs2: {fs2}\n")

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

n1 = sorted(nums)  # Numbers sort numerically by default
print("Numbers sorted numerically:")
print(f"n1: {n1}\n")

n2 = sorted(nums, key=str)  # Sort numbers as strings
print("Numbers sorted as strings:")
print(f"n2: {n2}\n")
```

custom_sort_keys.py

```
Ignoring case:
fs1: ['Apple', 'apricot', 'banana', 'BLUEberry', 'cherry', 'date', 'elderberry', 'FIG',
'grape', 'guava', 'Kiwi', 'lemon', 'lime', 'lychee', 'ORANGE', 'papaya', 'peach', 'pear',
'persimmon', 'pomegranate', 'Tamarind', 'Watermelon']

By length, then name:
fs2: ['FIG', 'date', 'Kiwi', 'lime', 'pear', 'Apple', 'grape', 'guava', 'lemon', 'peach',
'banana', 'cherry', 'lychee', 'ORANGE', 'papaya', 'apricot', 'Tamarind', 'BLUEberry',
'persimmon', 'elderberry', 'Watermelon', 'pomegranate']

Numbers sorted numerically:
n1: [5, 32, 80, 255, 400, 800, 1000, 5000]

Numbers sorted as strings:
n2: [1000, 255, 32, 400, 5, 5000, 80, 800]
```

Example

sort_holmes.py

```
"""Sort titles, ignoring leading articles"""
books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]


def strip_article(title):  # create function which takes element to compare and returns
comparison key
    title = title.lower()
    for article in 'a ', 'an ', 'the ':
        if title.startswith(article):
            title = title.removeprefix(article)  # remove article
            break
    return title


for book in sorted(books, key=strip_article):  # sort using custom function
    print(book)
```

sort_holmes.py

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```

Lambda functions

- Short cut function definition
- Useful for functions only used in one place
- Frequently passed as parameter to other functions
- Function body is an expression; it cannot contain other code

A **lambda function** is a brief function definition that makes it easy to create a function on the fly. This can be useful for passing functions into other functions, to be called later. Functions passed in this way are referred to as "callbacks". Normal functions can be callbacks as well. The advantage of a lambda function is solely the programmer’s convenience. There is no speed or other advantage.

One important use of lambda functions is for providing sort keys; another is to provide event handlers in GUI programming.

The basic syntax for creating a lambda function is

```
lambda parameter-list: expression
```

where parameter-list is a list of function parameters, and expression is an expression involving the parameters. The expression is the return value of the function.

A lambda function could also be defined in the normal manner

```
    def function-name(param-list):
        return expr
```

But it is not possible to use the normal syntax as a function parameter, or as an element in a list.

Example

lambda_example.py

```
fruits = ['watermelon', 'lime', 'Apple', 'Mango', 'KIWI', 'apricot', 'LEMON', 'guava']

sorted_fruits = sorted(fruits, key=lambda e: (len(e), e.lower()))  # The lambda function
takes one fruit name and returns a tuple containing the length of the name and the name
in lower case. This sorts first by length, then by name.

print(sorted_fruits)
```

lambda_example.py

```
['KIWI', 'lime', 'Apple', 'guava', 'LEMON', 'Mango', 'apricot', 'watermelon']
```

List comprehensions

- Filters or modifies elements
- Creates new list
- Shortcut for a for loop

A list comprehension is a Python idiom that creates a shortcut for a for loop. It returns a copy of a list with every element transformed via an expression. Functional programmers refer to this as a mapping function.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)   # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be added to filter values:

```
results = [ expr for var in sequence if expr ]
```

Example

listcomp.py

```
fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]

ufruits = [fruit.upper() for fruit in fruits]  # Copy each fruit to upper case

afruits = [fruit for fruit in fruits if fruit.startswith('a')]  # Select each fruit if it
starts with 'a'

doubles = [v * 2 for v in values]  # Copy each number, doubling it

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print()
```

listcomp.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: apple apricot
doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']
```

Dictionary comprehensions

- Expression is key/value pair
- Transform iterable to dictionary

A dictionary comprehension has syntax similar to a list comprehension. The expression is a key:value pair, and is added to the resulting dictionary. If a key is used more than once, it overrides any previous keys. This can be handy for building a dictionary from a sequence of values; another use is to modify the values in an existing dictionary.

Example

dict_comprehension.py

```
import os
from pprint import pprint

FOLDER = "../DATA"

file_names = 'alice.txt', 'parrot.txt', 'words.txt'

file_info = {name: os.path.getsize(os.path.join(FOLDER, name)) for name in file_names}

pprint(file_info)

print('-' * 60)

capitals = {'NY': 'ALBANY', 'NC': 'RALEIGH', 'CA': 'SACRAMENTO', 'VT': 'MONTPELIER'}

caps = {state: capital.title() for state, capital in capitals.items()}
pprint(caps)
```

dict_comprehension.py

```
{'alice.txt': 148544, 'parrot.txt': 1437, 'words.txt': 1749415}
------------------------------------------------------------
{'CA': 'Sacramento', 'NC': 'Raleigh', 'NY': 'Albany', 'VT': 'Montpelier'}
```

Set comprehensions

- Expression is added to set
- Transform iterable to set — with modifications

A set comprehension is useful for turning any sequence into a set. Items can be modified or skipped as the set is built.

If you don't need to modify the items, it's probably easier to just pass the sequence to the `set()` constructor.

Example

set_comprehension.py

```
import re

FILE_PATH = "../DATA/mary.txt"

# NOTE: r'\W+' is a regular expression that splits on anything that isn't a letter,
number, or underscore

with open(FILE_PATH) as mary_in:
    file_contents = mary_in.read()
    s = {w.lower() for w in re.split(r'\W+', file_contents) if w} # Get unique words from
file. Only one line is in memory at a time. Skip "empty" words.
print(s)
```

set_comprehension.py

```
{'the', 'white', 'lamb', 'a', 'little', 'fleece', 'go', 'as', 'went', 'snow', 'and',
'sure', 'its', 'had', 'to', 'that', 'mary', 'was', 'everywhere'}
```

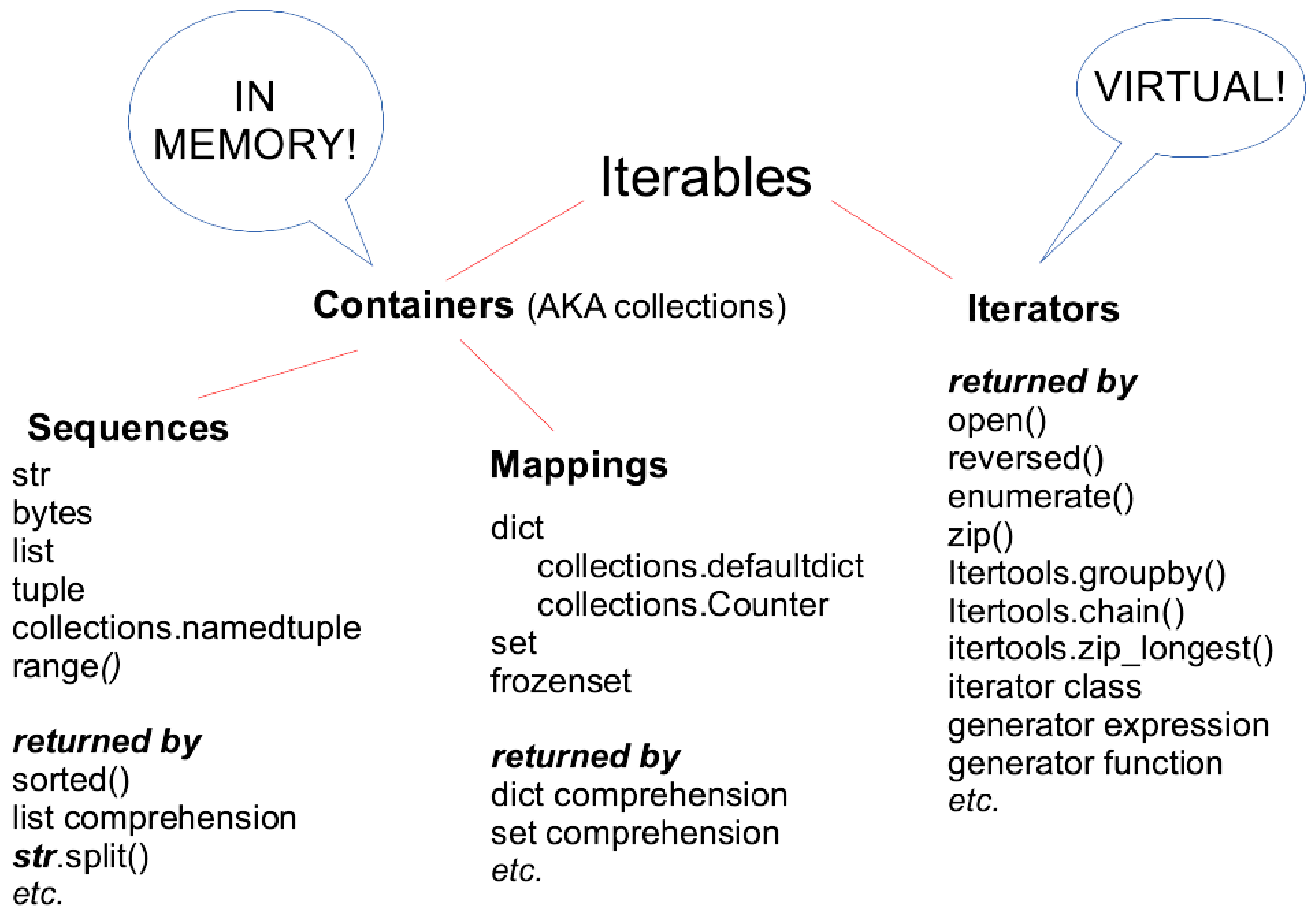
Iterables

- Expression that can be looped over
- Can be collections *e.g.* `list`, `tuple`, `str`, `bytes`
- Can be generators *e.g.* `range()`, file objects, `enumerate()`, `zip()`, `reversed()`

Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All builtin collections (`list`, `tuple`, `str`, `bytes`) are iterables. They keep all their values in memory. Many other builtin iterables are *generators*.

A generator does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for-in loop. This is a Good Thing, because it saves memory.



Generator Expressions

- Like list comprehensions, but create a generator object
- More efficient
- Use parentheses rather than brackets

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

Generator expressions are especially useful with functions like `sum()`, `min()`, and `max()` that reduce an iterable input to a single value:

There is an implied `yield` statement at the beginning of the expression.

Example

gen_ex.py

```
# sum the squares of a list of numbers
# using list comprehension, entire list is stored in memory
s1 = sum([x * x for x in range(10)])  # using list comprehension, entire list is stored
in memory

# only one square is in memory at a time with generator expression
s2 = sum(x * x for x in range(10))  # with generator expression, only one square is in
memory at a time
print(s1, s2)
print()
```

gen_ex.py

```
285 285
```

Generator functions

- Mostly like a normal function
- Use yield rather than return
- Maintains state

A generator is like a normal function, but instead of a return statement, it has a yield statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops. A generator function maintains state between calls, unlike a normal function.

Example

sieve_generator.py

```
def next_prime(limit):
    flags = set()  # initialize empty set (to be used for "is-prime" flags

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  # add non-prime elements to set
        yield i  # execution stops here until next value is requested by for-in loop


np = next_prime(200)  # next_prime() returns a generator object
for prime in np:  # iterate over yielded primes
    print(prime, end=' ')
```

sieve_generator.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

line_trimmer.py

```
def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            yield line.rstrip('\n\r')  # 'yield' causes this function to return a
generator object

mary_in = trimmed('../DATA/mary.txt')
for trimmed_line in mary_in:
    print(trimmed_line)
```

line_trimmer.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

Format strings

- Add `f` in front of literal strings
- Added in version 3.6
- Format: `{expression:formatting codes}`

The best way to format text in Python is *format strings*, better known as *f-strings*. An f-string is a literal string that starts with an `f` in front of the opening quotes. It can contain both literal text and *expressions*, which can be any normal Python expression.

Each expression is put inside curly braces (`{}`). This way, the value to be formatted is right where it will be in the resulting string. The expression can be any variable, as well as an expression or function call, such as `x + y` or `spam()`.

```
name = "Anna Karenina"
city = "Moscow"

s = f"{name} lives in {city}"
```

By default, the expression is converted to a string. You can add a colon and formatting codes to fine-tune how it is formatted.

```
result = 22 / 7
print(f"result is {result:.2f}")
```

To include literal braces in the string, double them: `{{ }}`.

See appendix [String Formatting] for details on formatting.

Format strings are fast and flexible, and a big improvement over `STR.format()`, which is described in the next section. They are evaluated at run time, so they are not constants. See file `f_strings_fun.py` for other examples.

For more details, check out the PyDoc topic FORMATTING, or section 6.1.3.1 of The Python Standard Library documentation, the **Format Specification Mini-Language**.

Example

f_strings.py

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print(f"It is {temperature}\u00B0 in {city}")
print()

# :03d means format (decimal) integer in 3 characters,
#      left-padded with zeros
# :.2f means round a float to 2 decimal points
print(f"hit count is {hit_count:03d} average is {average:.2f}")
print()

# any expression is OK
print(f"2 + 2 is {2 + 2}")
```

f_strings.py

```
It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4
```

Example

f_strings_fun.py

```
# fun with strings
name = "Guido"

print(f"name: {name}")
# < left justify (default for non-numbers), 10 is field width, s formats a string
print(f"name: [{name:<10s}]")
# > right justify
print(f"name: [{name:>10s}]")
# >. right justify and pad with dots
print(f"name: [{name:.>10s}]")
# ^ center
print(f"name: [{name:^10s}]")
# ^ center and pad with dashes
print(f"name: [{name:-^10s}]")
print()

# fun with integers
value = 2093
print(f"value: {value}")
print(f"value: [{value:10d}]")  # pad with spaces to width 10
print(f"value: [{value:010d}]")  # pad with zeroes to width 10
print(f"value: {value:d} {value:b} {value:x} {value:o}")  # d is decimal, b is binary, o
is octal, x is hex
print(f"value: {value} {value:#b} {value:#x} {value:#o}")  # add prefixes
print()

result = 38293892
print(f"result: ${result:,d}")  # , adds commas to numeric value
print()

# fun with floats
amount = .325482039
print(f"amount: {amount}")
print(f"amount: {amount:.2f}")  # round to 2 decimal places
print(f"amount: {amount:.2%}")  # convert to percent
print()

# fun with functions
print(f"length of 'name': {len(name)}")  # function call OK
```

f_strings_fun.py

```
name: Guido
name: [Guido     ]
name: [     Guido]
name: [.....Guido]
name: [  Guido   ]
name: [--Guido---]

value: 2093
value: [      2093]
value: [0000002093]
value: 2093 100000101101 82d 4055
value: 2093 0b100000101101 0x82d 0o4055

result: $38,293,892

amount: 0.325482039
amount: 0.33
amount: 32.55%

length of 'name': 5
```

Using the `.format()` method

- Expressions passed via `.format()`
- Same rules as `string.format()`

Prior to version 3.6, the best tool for formatted output was the `.format()` method on strings. This is very similar to f-strings, except that the expressions to be formatted are passed to the `.format()` method. Curly braces are used as before, but they are left empty. The first argument to `.format()` goes in the first pair of braces, etc.

Formatting codes still go after a colon.

There are many more ways of using `format()`; these are just some of the basics.

`.format()` is useful for reusing the same format with different values.

Example

string_formatting.py

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print("It is {}\u00B0 in {}".format(temperature, city))
print()

# :03d means format (decimal) integer in 3 characters,
#      left-padded with zeros
# :.2f means round a float to 2 decimal points
print("hit count is {:03d} average is {:.2f}".format(hit_count, average))
print()

# any expression is OK
print("2 + 2 is {}".format(2 + 2))
```

string_formatting.py

```
It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4
```

Legacy String Formatting

- Use the % operator
- Syntax: `"template" % (VALUES)`
- Similar to `printf()` in **C**

Prior to Python 2.6, when the `.format()` method was added to strings, the `%` symbol was used as a format operator. Like the more modern versions, this operator returns a string based on filling in a format string with values.

```
%flagW.Ptype
```

where W is width, P is precision (max width or # decimal places)

The placeholders are similar to those used in the C `print()` function. They are specified with a percent symbol (**%**), rather than braces.

If there is only one value to format, the value does not need parentheses.

Example

string_formatting_legacy.py

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print("It is %d\u00B0 in %s" % (temperature, city))
print()

# :03d means format (decimal) integer in 3 characters,
#      left-padded with zeros
# :.2f means round a float to 2 decimal points
print(f"hit count is %03d average is %.2f" % (hit_count, average))
print()

# any expression is OK
print(f"2 + 2 is %d" % (2 + 2))
```

string_formatting_legacy.py

```
It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4
```

Table 12. Legacy formatting types

placeholder	**data type**
d,i	decimal integer
o	octal integer
u	unsigned decimal integer
x,X	hex integer (lower, UPPER case)
e,E	scientific notation (lower, UPPER case)
f,F	floating point
g,G	autochoose between e and f
c	character
r	string (using repr() method)
s	string (using str() method)
%	literal percent sign

Table 13. Legacy formatting flags

flag	**description**
-	left justify (default is right justification)
#	use alternate format
0	left-pad number with zeros
+	precede number with + or -
(blank)	precede positive number with blank, negative with -

Chapter 6 Exercises

Exercise 6-1 (pres_upper.py)

Read the file `presidents.txt`, creating a list of of the presidents' last names. Then, use a list comprehension to make a copy of the list of names in upper case. Finally, loop through the list returned by the list comprehension and print out the names one per line.

Exercise 6-2 (pres_by_dob.py)

Print out all the presidents first and last names, date of birth, and their political affiliations, sorted by date of birth.

Read the `presidents.txt` file, putting the four fields into a list of tuples.

Loop through the list, sorting by date of birth, and printing the information for each president. Use `sorted()` and a lambda function.

Exercise 6-3 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME MIDDLENAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Then iterate over the the generator returned by your function and print the names.

Chapter 7: Packaging

Objectives

- Create a pyproject.toml file
- Understand the types of wheels
- Generate an installable wheel
- Configure dependencies
- Configure executable scripts
- Distribute and deploy packages

Packaging overview

- Bundling project for distribution
- Uses build tools
- Needs metadata
- Extremely flexible

Packaging a project for distribution does not have to be complex. However, the tools are very flexible, and the amount of configuration can be overwhelming at first.

It boils down to these steps:

Create a virtual environment

While not absolutely necessary, creating a virtual environment for your project makes life easier, especially when it comes to dependency management.

Create a project layout

Arrange files and subfolders in the project folder. This can be *src* (recommended) or *flat*.

Specify metadata

Using `pyproject.toml`, specify metadata for your project. This metadata tells the build tools how and where to build and package your project.

Build the project

Use the build tools to create a *wheel* file. This wheel file can be distributed to developers.

Install the wheel file

Anyone who wants to install your project can use `pip` to install the project from the wheel file you built.

Upload the project to PyPI (Optional)

To share a project with everyone, upload it to the **PyPI** online repository.

Terminology

Here are some terms used in Python packaging. They will be explained in more detail in the following pages.

build backend

A module that does the actual creation of installable files (wheels). E.g., `setuptools` (>=61), `poetry-core`, `hatchling`, `pdm-backend`, `flit-core`.

build frontend

A user interface for a build backend. E.g., `pip`, `build`, `poetry`, `hatch`, `pdm`, `flit`

cookiecutter

A tool to generate the files and folders needed for a project.

dependency

A package needed by the current package.

editable install

An installation that is really a link to the development folder, so changes to the code are reflected whenever and wherever the package or module is imported.

package

Can refer to either a **distribution package** or an **import package**.

distribution package

A collection of code (usually a folder) to be bundled into a reusable (installable) "artifact" AKA *wheel* file. A distribution package can be used to install **modules**, **import packages**, **scripts**, or any combination of those items.

```
pip install distribution-package
```

import package

An installable module, usually implemented as a folder that contains one or more module files.

```
import import_package
```

PEP

Python Enhancement Proposal — a document that describes some aspect of Python. Similar to RFCs in the Internet world.

pip

The standard tool to install a Python package.

script

An executable Python script that is installed in the `scripts` (Windows) or `bin` (Mac/Linux) folder of your Python installation

toml

A file format similar to INI that is used for describing projects.

wheel

A file that contains everything needed to install a package

Project layout

A typical project has several parts: source code, documentation, tests, and metadata. These can be laid out in different ways, but most people either do a *flat* layout or a *src* layout.

A *flat* layout has the code in the top level of the project, and a *src* layout has code in a separate folder named `src`.

In the long run, the *src* layout seems to be the most readable, and that makes it the best practice.

Metadata goes in the `pyproject.toml` file.

Unit tests go in a folder named `tests`. Documentation, using a tool such as **Sphinx**, goes in a folder named `docs`.

You can add any other files or folders necessary for your project.

Typical layout

```
temperature
 ├──── README.md
 ├──── docs
 │    ├──── Makefile
 │    ├──── make.bat
 │    └──── source
 │        ├──── _static
 │        ├──── _templates
 │        ├──── conf.py
 │        └──── index.rst
 ├──── pyproject.toml
 ├──── src
 │    └──── temperature.py
 └──── tests
     └──── test_temperature.py
```

the name of the project folder can be anything, but is typically the name of of the module or package you are creating.

The layouts on the following pages are not the only possibilities. You can combine them in whatever way works for your project.

Sample Project layouts

Module

```
MODULE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │    └──── mymodule.py
 └──── tests
```

Code in mymodule.py will run as follows:

```
# import module (in a script)
# __name__ set to "mymodule"
import mymodule
from mymodule import MyClass, myfunction

# execute module (from command line)
# __name__ set to "__main__"
python mymodule.py
python -m mymodule
```

Module with callable scripts

```
MODULE_SCRIPTS_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml  # script names mapped to mymodule:function
 ├──── src
 │    └──── mymodule.py  # functions called by scripts
|        |─── function1()
|        |─── _function1_wrapper()
 └──── tests
```

In `pyproject.toml`

```
[project.scripts]
myscript='mymodule:_function1_wrapper'
```

Module can be run or imported normally as above, plus scripts defined in `pyproject.toml` can run directly from the command line

```
myscript
```

Package

```
PACKAGE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │    └──── mypackage
 │        ├──── mymodule1.py
 │        └──── mymodule2.py
 └──── tests
```

```
from mypackage import mymodule1
from mypackage.mymodule1 import MyClass, myfunction
```

Package with subpackages

```
PACKAGE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │    └──── mypackage
 │        └──── mysubpackage1
 │            ├──── mymodule1.py
 │            └──── mymodule2.py
 │        └──── mysubpackage2
 │            ├──── mymodule3.py
 │            └──── mymodule4.py
 └──── tests
```

```
from mypackage.subpackage1 import mymodule1
from mypackage.subpackage2.mymodule3 import MyClass, myfunction
```

Package callable with `python -m packagename`

```
PACKAGE_APP_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│     └──── mypackage
│         ├──── __main__.py.py  # module executed by  python  -m packagename
│         ├──── mymodule1.py    # code to support main module
│         └──── mymodule2.py    # code to support main module
└──── tests
```

```
python -m mypackage
```

python -m *NAME*

- python -m module
  - executes entire module
- python -m package
  - executes module `__main__.py` in package
- includes `if __name__ == "__main__"` section

The command `python -m NAME` is designed to execute a module or package without knowing its exact location. It uses the module search mechanism to find and load the module. This searches the folders in `sys.path`.

If NAME is a package, it executes the module named `__main__.py` in the top level of the package.

If NAME is a module, it executes the entire module.

In both cases, the code in the `if __name__ == "__main__"` block (if any) is executed. If a module is imported, that code is not executed.

A list of standard modules that have a command line interface via `python -m` is here: https://docs.python.org/3/library/cmdline.html

Invoking Python

Assume the following code layout for the examples in the table

```
spam.py
ham
 ├──── __main__.py
 └──── toast.py
```

Invocation	Description	Example
*FILE and FOLDER specified as arguments to *python**		

`python FILE`	Run all code in `FILE`	`python spam.py`
		`python ham/toast.py`

`python FOLDER`	If `__main__.py` exists in `FOLDER`, run all code in `__main__.py`; otherwise, raise error	`python ham`
MODULE and PACKAGE found using sys.path		
`python -m MODULE`	Run all code in `MODULE`	`python -m spam`
`python -m PACKAGE`	Run all code in `PACKAGE.__init__.py` Run all code in `__main__`	`python -m ham`
`python -m PACKAGE.MODULE`	Run all code in `PACKAGE.__init__.py` Run all code in `PACKAGE.MODULE`	`python -m ham.toast`

Cookiecutter

- Creates standard layout
- Developed for Django
- Very flexible

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. However, it can be used create a layout for any type of project.

The `cookiecutter` command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are many templates on **github** to choose from, and you can easily create your own.

Syntax is

```
cookiecutter template-folder
```

The script copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places. It will do this in both file names and file contents.

There are two **cookiecutter** templates provided in the SETUP folder of the student files to generate the layouts on the previous pages:

- `cookiecutter-python-module`
- `cookiecutter-python-package`

The project layouts will be generated based on answers to the cookiecutter questions.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Feel free to copy the cookiecutter templates and modify them for your own projects.

Another useful tool for generating Python projects is **PyScaffold**. Details at https://pyscaffold.org/en/stable/index.html.

cookiecutter-python-package/cookiecutter.json

```
{
    "package_name": "Package Name (can have spaces)",
    "package_slug": "{{ cookiecutter.package_name.lower().replace(' ','').replace('-
','_') }}",
    "package_description": "Short Description of the Package",
    "module_slug": "{{ cookiecutter.package_slug }}",
    "has_scripts": "n",
    "has_main": "n",
    "author_name": "Author Name",
    "author_email": "noone@nowhere.com",
    "author_url": "Author URL",
    "copyright_year": "2024",
    "readme_format": ["md", "rst"]
}
```

tree cookiecutter-python-package

```
/Users/jstrick/curr/courses/python/common/setup/cookiecutter-python-package
├── {{cookiecutter.package_slug}}
│   ├── docs
│   │   ├── make.bat
│   │   ├── Makefile
│   │   └── source
│   │       ├── _static
│   │       ├── _templates
│   │       ├── conf.py
│   │       └── index.rst
│   ├── pyproject.toml
│   ├── README.{{cookiecutter.readme_format}}
│   ├── src
│   │   └── {{cookiecutter.package_slug}}
│   │       ├── __init__.py
│   │       ├── __main__.py
│   │       └── {{cookiecutter.module_slug}}.py
│   └── tests
│       ├── __init__.py
│       └── test_{{cookiecutter.module_slug}}.py
├── cookiecutter.json
└── hooks
    ├── post_gen_project.py
    └── pre_gen_project.py

9 directories, 14 files
```

cookiecutter cookiecutter-python-package

```
[1/11] package_name (Package Name (can have spaces)): Log Processor
[2/11] package_slug (logprocessor): logproc
[3/11] package_description (Short Description of the Package): Process Log Files
[4/11] module_slug (logproc):
[5/11] has_scripts (n): y
[6/11] has_main (n): y
[7/11] author_name (Author Name): Sabrina Q. Programmer
[8/11] author_email (noone@nowhere.com): sabrinaq@gmail.com
[9/11] author_url (Author URL): https://www.sabrinaq.com
[10/11] copyright_year (2024):
[11/11] Select readme_format
1 - md
2 - rst
Choose from [1/2] (1): 1
```

tree logproc

```
logproc
├── README.md
├── docs
│   ├── Makefile
│   ├── make.bat
│   └── source
│       ├── _static
│       ├── _templates
│       ├── conf.py
│       └── index.rst
├── pyproject.toml
├── src
│   ├── logproc
│       ├── __init__.py
│       ├── __main__.py
│       └── logproc.py
└── tests
    ├── __init__.py
    └── test_logproc.py
```

Defining project metadata

- Create `pyproject.toml`
- Use `build` to build the package

The **modern** way to package a Python project is using the `pyproject.toml` config file. The specifications that support this are specified in **PEP 518** and **PEP 621**.

The **TOML** format is similar to `.ini` files, but adds some features.

The first part of the file is always required. It tells the `build` program what tools to use.

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

Put all the project metadata that the build system will need to package and install your project after the `[build-system]` section.

```
[project]
name = "logproc"
version = "1.0.0"
authors = [
    { name="Author Name", email="sabrinaq@gmail.com" },
]
description = "Short Description of the Package"
readme = "README.rst"
requires-python = ">=3.0"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

dependencies = [
    'requests[security] < 3',
]
```

TOML value types arrays are similar to Python `list` and TOML tables (including inline) are similar to `dict`.

The rest of the file after the `[build-system]` section is not needed if you are also using `setup.cfg` and `setup.py`. However, best practice is to *not* use those legacy files, as all the data needed for building the package, installing it, and uploading it to **PyPI** can be contained in `pyproject.toml`, and can then be used by nearly any build *backend*.

Packages with scripts

- Provide utility scripts
- Run from command line
- Installed in `…/scripts` or `…/bin`
- Add config to `pyproject.toml`

It is easy to add one or more command-line scripts to your project. These scripts are created in the `scripts` (Windows) or `bin` (other OS) folders of your Python installation. While they require Python to be installed, they are run like any other command.

The scripts are based on functions in the module. Since the scripts are run from the CLI, they are not called with normal parameters. Instead, the functions access `sys.argv` for arguments, like any standalone Python script.

```
def _c2f_cli():
    """
    CLI utility script
    Called from command line as 'c2f'
    """
    cel = float(sys.argv[1])
    return c2f(cel)


def _f2c_cli():
    """
    CLI utility script
    Called from command line as 'f2c'
    """
    fahr = float(sys.argv[1])
    return f2c(fahr)
```

To configure scripts, add a section like the following to `pyproject.toml`. The names on the left are the installed script names. The values on the right are the module name and the function name, separated by a colon.

```
[project.scripts]
c2f = 'temperature:_c2f_cli'
f2c = 'temperature:_f2c_cli'
```

See the project `temperature_scripts` in EXAMPLES for details.

Editable installs

- Use `pip install -e package`
- Puts a link in library folder
- Allows testing as though module is installed

When using a `src` (or other name) folder for your codebase and `tests` for your test scripts, the tests need to find your package. While you could put the path to the `src` folder in PYTHONPATH, the best practice is to do an *editable install*.

This is an install that uses the path to your development folder. It achieves this by using a virtual environment. Then you can run your tests after making changes to your code, without having to reinstall the package.

From the top level folder of the project, type the following (you do not have to build the distribution for this step).

```
pip install -e .
```

Now the project is installed and is available to import or run like any other installed module.

Running unit tests

- Use editable install
- Just use `pytest` or `pytest -v`

To run the tests that you have created in the `tests` folder, just run `pytest` or `pytest -v` (verbose) in the top level folder of the project. Because the project was installed with an editable install, tests can import the module or package normally.

Example

```
$ pytest -v
=========================================================== test session starts
============================================================
platform darwin -- Python 3.9.17, pytest-7.1.2, pluggy-1.0.0 -- /Users/jstrick/opt/miniconda3/bin/python
cachedir: .pytest_cache
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/common/examples/temperature/.hypothesis/examples')
rootdir: /Users/jstrick/curr/courses/python/common/examples/temperature
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0, astropy-header-0.2.1, fixture-order-
0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3,
openfiles-0.5.0, django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 8 items

tests/test_temperature.py::test_c2f[100-212] PASSED
[ 12%]
tests/test_temperature.py::test_c2f[0-32] PASSED
[ 25%]
tests/test_temperature.py::test_c2f[37-98.6] PASSED
[ 37%]
tests/test_temperature.py::test_c2f[-40--40] PASSED
[ 50%]
tests/test_temperature.py::test_f2c[212-100] PASSED
[ 62%]
tests/test_temperature.py::test_f2c[32-0] PASSED
[ 75%]
tests/test_temperature.py::test_f2c[98.6-37] PASSED
[ 87%]
tests/test_temperature.py::test_f2c[-40--40] PASSED
[100%]

============================================================ 8 passed in 0.20s
=============================================================
```

Wheels

- 3 kinds of wheels
    - Universal wheels (pure Python; python 2 *and* 3 compatible
    - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
    - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A *Universal wheel* is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A *Pure Python wheel* is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A *Platform wheel* is a package that has extensions, and thus is platform-specific.

Build systems automatically create the correct wheel type.

Building distributions

- `python -m build`
- Creates `dist` folder
- Binary distribution
  - `package-version.whl`
- Source distribution
  - `package-version.tar.gz`

To build the project, use

```
python -m build
```

This will create the wheel file (binary distribution) and a gzipped tar file (source distribution) in a folder named `dist`.

python -m build

```
* Creating virtualenv isolated environment...
* Installing packages in isolated environment... (setuptools>=61.0)
* Getting build dependencies for sdist...
running egg_info
writing src/temperature.egg-info/PKG-INFO
writing dependency_links to src/temperature.egg-info/dependency_links.txt
writing top-level names to src/temperature.egg-info/top_level.txt
reading manifest file 'src/temperature.egg-info/SOURCES.txt'
writing manifest file 'src/temperature.egg-info/SOURCES.txt'
* Building sdist...
```

... about 70 lines of output ...

```
running install_scripts
creating build/bdist.macosx-10.9-universal2/wheel/temperature-1.0.0.dist-info/WHEEL
creating '/Users/jstrick/curr/courses/python/common/examples/temperature/dist/.tmp-
e6gqm7bb/temperature-1.0.0-py3-none-any.whl' and adding 'build/bdist.macosx-10.9-
universal2/wheel' to it
adding 'temperature.py'
adding 'temperature-1.0.0.dist-info/METADATA'
adding 'temperature-1.0.0.dist-info/WHEEL'
adding 'temperature-1.0.0.dist-info/top_level.txt'
adding 'temperature-1.0.0.dist-info/RECORD'
removing build/bdist.macosx-10.9-universal2/wheel
Successfully built temperature-1.0.0.tar.gz and temperature-1.0.0-py3-none-any.whl
```

Installing a package

- Use `pip`
  - many options
  - can install just for user

A wheel makes installing packages simple. You can just use

```
pip install package.whl
```

This will install the package in the standard location for the current version of Python.

If you do not have permission to install modules in the standard location, you can do a user install, which installs modules under your home folder.

```
pip install --user package.whl
```

For more information

Python Packaging User Guide

https://packaging.python.org/en/latest/

Distributing Python Modules

https://docs.python.org/3/distributing/index.html

setuptools Quickstart

https://setuptools.pypa.io/en/latest/userguide/quickstart.html

Thoughts on the Python packaging ecosystem

https://pradyunsg.me/blog/2023/01/21/thoughts-on-python-packaging/

THE BASICS OF PYTHON PACKAGING IN EARLY 2023

https://drivendata.co/blog/python-packaging-2023

Structuring Your Project (from The Hitchhiker's Guide to Python)

https://docs.python-guide.org/writing/structure/

Chapter 7 Exercises

Exercise 7-1 (carddeck/*)

Step 1

Create a distributable module named `carddeck` from the `carddeck.py` and `card.py` modules in the root folder of the student files.

HINT: To do it the easy way, use the `cookiecutter-python-module` template. Add the two source files to the `src` folder. (remove any existing sample Python scripts in `src`).

To do it the "hard" way, create the project layout by hand, create the `pyproject.toml` file, etc.

Step 2

Build a distribution (wheel file).

Step 3

Install the wheel file with `pip`. (The cookiecutter template automatically does an editable install)

Step 4

Then import the new module and create an instance of the `CardDeck` class. Shuffle the cards, and deal out all 52 cards.

Chapter 8: Developer Tools

Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

Program development

- More than just coding
    - Design first
    - Consistent style
    - Comments
    - Debugging
    - Testing
    - Documentation

Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don't state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1       # Increment x
```

Only use an inline comment if the reason for the statement is not obvious:

```
x = x + 1       # Add one so range() does the right thing
```

The above was adapted from PEP 8

pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don’t worry if your code has a lower rating!
- Can be highly customized

pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler’s Refactoring book)

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab http://www.logilab.fr.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions.

pylint usage:

```
pylint filename(s) or directory
pylint -ry filename(s) or directory
```

The **-ry** option says to generate a full detailed report.

Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code:

- pyflakes
- pychecker

Customizing pylint

- Use pylint --generate-rcfile
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Customize
  - Use `▯-generate-rcfile` to generate default config
  - Redirect output to .pylintrc
  - Put in home folder

To customize pylint, run pylint with only the `-generate-rcfile` option. This will output a well-commented configuration file to STDOUT, so redirect it to a file named `.pylintrc`. Put the file in your home folder (%USERPROFILE% on Windows).

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

pylint will also find look for a file named `pylintrc` in the current directory, and on non-Windows systesm, `/etc/pylintrc`.

See https://docs.pylint.org for more details.

Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of `pylint`
- Generates UML diagrams

pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the `pylint` package.

There are many options to control what it analyzes and what kind of output it produces. See the output of `pyreverse -h` for all options.

Common pyreverse options

`-A` search all ancestors
`-p` specify project name
`-o` specify output type (e.g., HTML)
`-d` specify output folder `-h` show help

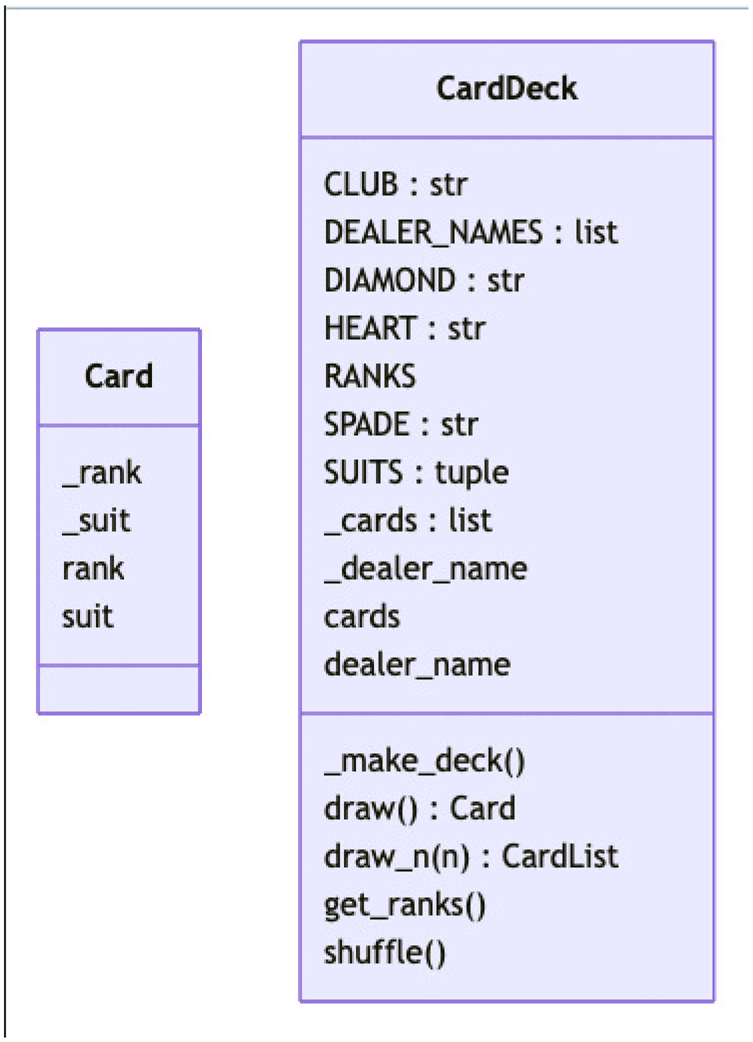
Example

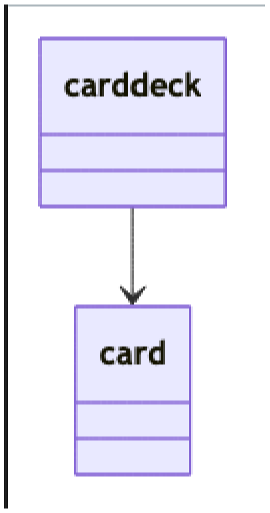
```
$ pyreverse -p carddeck -f SPECIAL -S  -A -o html --colorized card carddeck jokerdeck
```

Table 14. Builtin pyreverse output types

Format	Description	Notes
html	Web page	Open in browser
dot	Graph description Language	Create images with **Graphviz**
puml plantuml	Plant UML	Create images with **plantuml** Java app
mmd	Mermaid	Create image with Mermaid command line tool

For images in standard formats such as **pdf** or **png**, `pyreverse` requires **Graphviz**, a graphics tool that must be installed separately from Python. `pyreverse` will generate a **dot** file and then use Graphviz to create the image. See https://graphviz.org/docs/outputs/ for a list of all Graphviz output types.


Card
_rank
_suit
rank
suit
CardDeck
CLUB : str
DEALER_NAMES : list
DIAMOND : str
HEART : str
RANKS
SPADE : str
SUITS : tuple
_cards : list
_dealer_name
cards
dealer_name
_make_deck()
draw() : Card
draw_n(n) : CardList
get_ranks()
shuffle()


carddeck
card

The Python debugger

- Implemented via `pdb` module
- Supports breakpoints and single stepping
- Based on **gdb**

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The `pdb` module provides debugging facilities for Python.

The usual way to use `pdb` is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- `python -m pdb script`

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Table 15. Common debugging commands

Command	Description
`n`	execute next line, stepping over functions
`s`	execute next line, stepping over functions
`b n`	set breakpoint at line n
`b function`	set breakpoint at function
`c`	continue to next breakpoint
`r`	return from function
`h`	show help

To get more help, type `h` at the debugger prompt.

Typical usage

```
python -m pdb play_cards.py
> /Users/jstrick/curr/courses/python/common/examples/play_cards.py(1)<module>()
→ from carddeck import CardDeck
(Pdb) l
  1  →    from carddeck import CardDeck
  2
  3     deck = CardDeck("Mary")
  4
  5     deck.shuffle()
  6
  7     for _ in range(10):
  8         card = deck.draw()
  9         print(card)
 10     print()
 11
(Pdb) b 5
(Pdb) b CardDeck.draw
Breakpoint 1 at /Users/jstrick/curr/courses/python/common/examples/play_cards.py:5
(Pdb) c
> /Users/jstrick/curr/courses/python/common/examples/play_cards.py(5)<module>()
→ deck.shuffle()
(Pdb) s
--Call--
> /Users/jstrick/curr/courses/python/common/examples/carddeck.py(51)shuffle()
→ def shuffle(self):
(Pdb) n
> /Users/jstrick/curr/courses/python/common/examples/carddeck.py(57)shuffle()
→ random.shuffle(self._cards)
(Pdb) p self._cards
```

Stepping through a program

- `s` single-step, stepping into functions
- `n` single-step, stepping over functions
- `r` return from function
- `c` run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use `s` to step through one line at a time, stepping into functions.

Use `n` to step over functions; use `r` to return from a function; use `c` to continue to next breakpoint or end of program.

Pressing `ENTER` repeats most commands; if the previous command was `l` (list), the debugger lists the next set of lines.

Setting breakpoints

```
b b linenumber (, condition) b file:linenumber (, condition) b function name (, condition)
```

Breakpoints can be set with the b command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The `tbreak` command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the `profile` module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

The `profile` utility reports statistics on all function calls during the execution of your program.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT.

To order by a specific column, use the `-s` option with any of the column names.

```
python -m profile -s tottime scriptname.py
```

You can specify an output file with the `-o` option, which can be then be examined with the `pstats` utility.

See https://docs.python.org/3/library/profile.html for more information.

The `pycallgraph2` module (third-party module) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

Example

```
python -m profile count_with_dict.py
...script output...
         19 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       14    0.000    0.000    0.000    0.000 :0(get)
        1    0.000    0.000    0.000    0.000 :0(items)
        1    0.000    0.000    0.000    0.000 :0(open)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 count_with_dict.py:3(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(<code object <module> at
0xb74c36e0, file "count_with_dict.py", line 3>)
        0    0.000             0.000          profile:0(profiler)
```

Benchmarking

- Use the `timeit` module
- Create a `Timer` object with specified # of repetitions

Use the `timeit` module to benchmark two or more code snippets. To time code, create a `Timer` object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the `.timeit()` method with the number of times to call the test code, or call the `.repeat()` method which repeats `.timeit()` a specified number of times.

You can also use `timeit` from the command line. Use `-s` to specify startup code:

```
python -m timeit -s ▯startup code⋯▯ ▯code⋯.▯
```

Example

bm_range_vs_while.py

```
from timeit import Timer

REPEATS = 10000

setup = """
values = []
"""  # setup code is only executed once

code_snippets = [
'''
for i in range(10000):
    values.append(i)
values.clear()
''',  # code fragment executed many times
'''
i = 0
while i < 10000:
    values.append(i)
    i += 1
values.clear()
''',  # code fragment executed many times
]

for code_snippet in code_snippets:
    t = Timer(code_snippet, setup)
    print(f"{code_snippet:80.80s}{t.timeit(REPEATS)}")
    print('-' * 60)
```

bm_range_vs_while.py

```
for i in range(10000):
    values.append(i)
values.clear()
                    2.783242107980186
------------------------------------------------------------


i = 0
while i < 10000:
    values.append(i)
    i += 1
values.clear()
         3.6577929769991897
------------------------------------------------------------
```

For more information

- https://pylint.readthedocs.io/en/latest/index.html
- https://graphviz.org/
- https://mermaid.js.org/
- https://docs.python.org/3/library/profile.html
- https://web.archive.org/web/20170318204046/http://lanyrd.com/2013/pycon/scdywg/

Chapter 8 Exercises

Exercise 8-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 8-2

Use the builtin debugger or one included with your IDE to step through any of the scripts you have written so far.

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

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

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

```python
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 123145454272512 with value apple
Starting thread 123145521430528 with value papaya
Starting thread 123145538220032 with value cherry
Starting thread 123145471062016 with value banana
Starting thread 123145504641024 with value peach
Starting thread 123145487851520 with value mango
Starting thread 123145555009536 with value lemon
Starting thread 123145571799040 with value watermelon
['APPLE', 'PAPAYA', 'CHERRY', 'BANANA', 'PEACH', 'MANGO', 'LEMON', '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/B1A11B2A12B3B4A13B5A14B6A15/clearing/A16A17A18A19
A20/setting/B7A21B8A22B9A23B10B11A24B12A25/clearing/A26A27A28A29A30/setting/B13A31B14A32B
15A33B16A34B17A35/clearing/B18A36A37A38A39A40/setting/B19A41B20A42B21A43B22A44B23A45/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

```
['NONSOCIAL-NONSOCIAL', 'WITHERITE-WITHERITE', 'MIDSPACE-MIDSPACE', 'AVADAVAT-AVADAVAT',
'DEFAME-DEFAME', 'COOKY-COOKY', 'ISOGONIES-ISOGONIES', 'OUTFROWNING-OUTFROWNING',
'SWIRLED-SWIRLED', 'ECTODERM-ECTODERM', 'ADDICTED-ADDICTED', 'INGLENOOKS-INGLENOOKS',
'INTERCASTE-INTERCASTE', 'BETAKING-BETAKING', 'SNIFFISHLY-SNIFFISHLY', 'CLERKDOM-
CLERKDOM', 'DESTITUTION-DESTITUTION', 'HARMIN-HARMIN', 'SPINDLED-SPINDLED', 'CIRCUITRY-
CIRCUITRY']
Fri Sep 26 13:52:13 2025
Fri Sep 26 13:52:13 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

```
['RESODDED', 'FLUORESCES', 'QUAKIER', 'MALARKIES', 'TIERED', 'MODISTES', 'VITALISTIC',
'MANEUVER', 'SOVIETIZING', 'WUSSES', 'BEAUTEOUSLY', 'APPELLATE', 'SHOVELS', 'LANGSHANS',
'GARBLESS', 'BIOCONTROL', 'REPUBLISH', 'REEMIT', 'BAZOOKA', 'CHILDLIKENESS', 'DELIRIUM',
'INTROVERSIVELY', 'SPUNKIEST', 'FLAGELLATE', 'SCEPTRAL', 'DOYLEY', 'SPOOKISH', 'HEMIN',
'FLEABAG', 'NEPHROSIS', 'SUBROGATE', 'MOSEYED', 'COMMA', 'PLASMAGENES', 'DEOXIDIZE',
'ULTRAPARADOXICAL', 'WINDUP', 'TSUNAMIS', 'MELANOTIC', 'HISTORICITY', 'AEROBICS',
'PHYSIOLOGIES', 'GLUTS', 'PRAISEWORTHINESS', 'SALES', 'LITTER', 'MISQUOTED', 'FRUITERS',
'SOUDAN', 'MORAINE']
25000
Fri Sep 26 13:52:13 2025
Fri Sep 26 13:52:17 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

```
['ROTATABLE', 'INCESTUOUSNESS', 'REACCELERATES', 'CONTORTIVE', 'LEAGUING', 'TYMPAN',
'SUZERAINS', 'ICTERUS', 'SWAMPIEST', 'DELEGACY', 'OBOVOID', 'TAMBURAS', 'UNINTERRUPTED',
'TRANSSONIC', 'SEXPLOITATION', 'VOYAGER', 'VANDA', 'HONEYMOON', 'CAJAPUT', 'LOLLS']
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 asynchronusly

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 16. 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 9 Exercises

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

Exercise 9-1 (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 9-2 (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()**.

Available topics if time permits

If there is time available at the end of class, any of the following topics may be covered based on student interest, and at the discretion of the instructor.

Chapter 10: Virtual Environments

Objectives

- Learn what virtual environments are
- Understand why you should use them
- Implement virtual environments

Why do we need virtual environments?

Consider the following scenario:

Mary creates an app using Python modules **spamlib** and **hamlib**. She builds a distribution package and gives it to Paul. He installs **spamlib** and **hamlib** on his computer, which has a compatible Python interpreter, and then installs Mary’s app. It starts to run, but then crashes with errors. What happened?

In the meantime, since Mary created her app, the author of **spamlib** removed a function "that almost noone used". When Paul installed **spamlib**, he installed the latest version, which does not have the function. One possible fix is for Paul to revert to an older version of **spamlib**, but suppose he has another app that uses the newer version? This can get very messy very quickly.

The solution to this problem is a **virtual environment**.

What are virtual environments?

A virtual environment starts with a snapshot (copy) of a plain Python installation, before any other libraries are added. It is used to isolate a particular set of modules that will successfully run a given application.

Each application can have its own virtual environment, which ensures that it has the required versions of dependencies. Virtual environments do not have to be in the same folder or folder tree as projects that use them, and multiple projects can use the same virtual environment. However, it’s best practice for each project to have its own.

There are many tools for creating and using virtual environments, but the primary ones are **pip** and the **venv** module.

Creating the virtual environment

Before you create the first environment

Before creating a virtual environment, it is best to start with a "plain vanilla" Python installation of the desired release. You can use the base Python bundle from https://www.python.org/downloads, or the Anaconda bundle from https://www.anaconda.com/distribution.

You won’t use this installation directly — it will be more a "base", or "reference" installation.

Creating the environment

The **venv** module provides the tools to create a new virtual environment. Basic usage is

```
python -m venv environment_name
```

This creates a virtual environment named *environment_name* in the current directory. It is a copy of the required parts of the original installation.

The virtual environment does not need to in the same location as your application. It is common to create a folder named *.envs* under your home folder to contain all your virtual environments.

Activating the environment

To use a virtual environment, it must be **activated**. This means that it takes precedence over any other installed Python version. This is is implemented by changing the PATH variable in your operating system's environment to point to the virtual copy.

venv will put the name of the environment in the terminal prompt.

Activating on Windows

To **activate** the environment on a Windows system, run the **activate.bat** script in the **Scripts** folder of the environment.

```
path-to-environment\Scripts\activate
```

Activating on non-Windows

To activate the environment on a Linux, Mac, or other Unix-like system, source the **activate** script in the **bin** folder of the environment. This must be *sourced* — run the script with the **source** builtin command, or the **.** shortcut

```
source path-to-environment/bin/activate
or
. path-to-environment/bin/activate
```

Deactivating the environment

When you are finished with an environment, you can deactivate it with the **deactivate** command. It does not need the leading path.

The command is the same on both Windows and non-Windows systems.

```
deactivate
```

venv will remove the name of the environment from the terminal prompt.

To run an app with a particular environment, create a batch file or shell script that activates the environment, then runs the app with the environment's interpreter. When the script is finished, it should deactivate the environment.

Freezing the environment

When ready to share your app, create a list of your project's dependencies in a file named `requirements.txt`. Then you can provide the file as part of your installation package. Users of your project can create a Python environment with the same versions of all required modules.

Run `pip freeze > requirements.txt` . This will create a list of all the modules you have added to the environment, with their versions.

Example

requirements.txt

```
ansi==0.3.7
beautifulsoup4==4.12.2
build==0.10.0
cookiecutter==2.4.0
cx-Oracle==8.3.0
Django==4.2.6
GitPython==3.1.40
h5py==3.10.0
ipython==8.16.1
Jinja2==3.1.2
jupyter==1.0.0
jupyterlab==4.0.7
lxml==4.9.3
matplotlib==3.8.0
memory-profiler==0.61.0
mypy==1.6.1
netmiko==4.2.0
numba==0.60.0
numpy==1.26.1
openpyxl==3.1.2
pandas==2.1.1
paramiko==3.3.1
psycopg==3.2.1
pylint==3.3.1
PyMySQL==1.1.0
pytest==7.4.2
PyYAML==6.0.1
requests==2.31.0
scikit-learn==1.3.2
scipy==1.11.3
seaborn==0.13.0
sympy==1.12
```

Using Conda

If you are using the Anaconda Distribution, it provides the `conda` tool. This can be used for installing packages as well as virtual environment management. (I.e., it replaces both `venv` and `pip`) The steps are similar to using `venv`, with a few extra conveniences.

One difference is that all environments are stored in a folder named `venvs` under the global Anaconda installation folder.

You can discover the installation folder by importing the `sys` module and printing the value of `sys.prefix`

Creating an environment

To create a new environment, use `conda create -n ENV_NAME`. This will create the new environment in `ANACONDA_FOLDER/envs/ENV_NAME`.

You can specify a particular version of Python with `conda create -n ENV_NAME python=X.y`

Activating the environment

Use `conda activate ENV_NAME`. You do not have to specify the path to the environment.

Use `conda info -e` to list all your available environments.

Deactivating the environment

Use `conda deactivate`

Installing modules

You can add modules to the environment with `conda install -n ENV_NAME PACKAGE …`

Freezing the environment

Run `conda list --export > requirements.txt`. Like `pip freeze`, this will create a list of the modules you have added to the environment.

Duplicating an environment

When someone sends you an app with a `requirements.txt` file, it is easy to reproduce their environments. First install a base version of Python, Then install dependencies (packages) contained in `requirements.txt`.

This will add the module dependencies with the correct versions.

Using pip

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

Using Conda:

```
conda install --yes --file requirements.txt
```

The `--yes` option automatically answers "yes" to the confirmation prompts.

The virtual enviroment swamp

There are many tools for virtual environment management, and having them all available can be confusing. You can always just use `pip` and `venv`, or `conda`, as described above.

Here are a few of these tools, and what they do

`pipenv`

A convenience tool that replaces both pip and venv.

`virtualenv`

The original name of the **venv** module.

Visual Studio Code, PyCharm

Python IDEs that will create virtual environments for you. They do not come with Python itself — that will have to be installed separately.

`virtualenvwrapper`

A workflow manager that makes it more convenient to switch from one environment to another.

Chapter 10 Exercises

Exercise 10-1

Create a virtual environment named **spam** and activate it.

Install the **roman** module.

Create a **requirements.txt** file.

Deactivate **spam**.

Create another virtual environment named **ham** and activate it.

Using **requirements.txt**, make the **ham** environment the same as **spam**.

Chapter 11: Advanced Data Handling

Objectives

- Set default dictionary values
- Count items with the Counter object
- Define named tuples
- Prettyprint data structures
- Create and extract from compressed archives
- Save Python structures to the hard drive

Deep vs shallow copying

- Normal assignments create aliases
- New objects are shallow copies
- Use the `copy.deepcopy` module for deep copies *

Consider the following code:

```
colors = ['red', 'blue', 'green']
c1 = colors
```

The assignment to variable `c1` does not create a new object; `c1` is another name that is *bound* to the same list object as the variable `colors`.

To create a new object, you can either use the list constructor list(), or use a slice which contains all elements:

```
c2 = list(colors)
c3 = colors[::]
```

In both cases, `c2` and `c3` are each distinct objects.

However, the elements of `c2` and `c3` are not copied, but are still bound to the same objects as the elements of `colors`.

For example:

```
data1 = [ [1, 2, 3], [4, 5, 6] ]
d1 = list(data)
d1[0].append(99)
print(data1)
```

This will show that the first element of data1 contains the value 99, because `data1[0]` and `d1[0]` are both bound to the same object. To do a *deep* (recursive) copy, use the `deepcopy` function in the `copy` module.

Example

deep_copy.py

```
import copy

data = [
    [1, 2, 3],
    [4, 5, 6],
]

d1 = data # Bind d1 to same object as data
d2 = list(data) # Make shallow copy of data and store in d2
d3 = copy.deepcopy(data) # Make deep copy of data and store in d3

d1.append("d1")  # Append to d1 (same as appending to data)
d1[0].append(50) # Append to first element of d1 (same first element as data)

d2.append("d2")   # Append to d2 (does not affect data)
d2[0].append(99)  # Append to first element of d2 (same first element as data and d1)

d3.append("d3")   # Append to d3 (does not affect data)
d3[0].append(500) # Append to first element of d3 (does not affect data, d1, or d2)

print("data:", data, id(data))
print("d1:", d1, id(d1))
print("d2:", d2, id(d2))
print("d3:", d3, id(d3))
print()

print("id(d1[0]):", id(d1[0]))
print("id(d2[0]):", id(d2[0]))
print("id(d3[0]):", id(d3[0]))
```

deep_copy.py

```
data: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4410014528
d1: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4410014528
d2: [[1, 2, 3, 50, 99], [4, 5, 6], 'd2'] 4410014656
d3: [[1, 2, 3, 500], [4, 5, 6], 'd3'] 4410014592

id(d1[0]): 4410014464
id(d2[0]): 4410014464
id(d3[0]): 4410014400
```

Default dictionary values

- Use `defaultdict` from the `collections` module
- Specify function that provides default value
- Good for counting, datasets

Normally, when you use an invalid key with a dictionary, it raises a `KeyError`. The `defaultdict` class in the `collections` module allows you to provide a default value, so there will never be a `KeyError`. You will provide a function that returns the default value. A lambda function can be used for the function.

Example

defaultdict_ex.py

```
from collections import defaultdict

dd = defaultdict(lambda: 0)  # create default dict with function that returns 0

dd['spam'] = 10  # assign some values to the dict
dd['eggs'] = 22

print(dd['spam'])  # print values
print(dd['eggs'])
print(dd['foo'])  # missing key 'foo' invokes function and returns 0

print('-' * 60)

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

fruit_info = defaultdict(list)

for fruit in fruits:
    first_letter = fruit[0]
    fruit_info[first_letter].append(fruit)

for letter, fruits in sorted(fruit_info.items()):
    print(letter, fruits)
```

defaultdict_ex.py

```
10
22
0
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
```

Counting with Counter

- Use `collections.Counter`
- Default value is 0
- Initialize or increment elements

For ease in counting, `collections` provides a `Counter` object. This is essentially a `defaultdict` whose default value is zero. Thus, you can just increment the value for any key, whether it's been seen before or no.

In addition, you can initialize a `Counter` object with any iterable.

The `pandas` module will also count data from datasets.

Example

count_with_counter.py

```
from collections import Counter

with open("../DATA/breakfast.txt") as breakfast_in:
    foods = [line.rstrip() for line in breakfast_in]  # create list of foods with EOL
removed from line

counts = Counter(foods)  # initialize Counter object with list of foods

for item, count in counts.items():  # iterate over results
    print(item, count)
print()

print(f"{counts.most_common(4) = }")
```

count_with_counter.py

```
spam 34
Lucky Charms 4
eggs 3
oatmeal 1
sausage 4
upma 3
poha 4
ackee and saltfish 4
bacon 6
pancakes 2
idli 7
dosas 4
waffles 2
crumpets 1

counts.most_common(4) = [('spam', 34), ('idli', 7), ('bacon', 6), ('Lucky Charms', 4)]
```

Named Tuples

- In `collections` module
- Like `tuple`, but each element has a name
- Use either one
  - `tuple.name`
  - `tuple[n]`

A *named tuple* is a tuple where each element has a name, and can be accessed via the name in addition to the normal access by index. Thus, if `p` were a named tuple representing a point, you could say `p.x` and `p.y`, or you could say `p[0]` and `p[1]`.

A named tuple is created with the `namedtuple` class in the `collections` module. You are essentially creating a new class that inherits from `tuple`.

Pass the name of the new named tuple followed by a string containing the individual field names separated by spaces. `namedtuple` returns a new class containing the specified fields. It must be initialized with values for all fields. Being a tuple, it may not be altered once created.

Convert a named tuple into a dictionary with the `._asdict()` method.

A named tuple is the closest thing Python has to a C struct.

Example

named_tuples.py

```
from collections import namedtuple
from pprint import pprint

Knight = namedtuple('Knight', 'name title color quest comment') # create named tuple
class with specified fields (could also provide fieldnames as iterable)

k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?') # create named tuple
instance (must specify all fields)

print(k.title, k.name) # can access fields by name...
print(k[1], k[0]) # ...or index
print()

knights = [] # initialize list for knight data
with open('../DATA/knights.txt') as knights_in:
    for raw_line in knights_in:
        # strip \n then split line into fields
        name, title, color, quest, comment = raw_line.rstrip().split(':')
        # create instance of Knight namedtuple
        knight = Knight(name, title, color, quest, comment)
        # add tuple to list
        knights.append(knight)

for knight in knights: # iterate over list of knights
    print(f'{knight.title} {knight.name}: {knight.color}')
print()
pprint(knights)
```

named_tuples.py

```
Sir Bob
Sir Bob

King Arthur: blue
Sir Galahad: red
Sir Lancelot: blue
Sir Robin: yellow
Sir Bedevere: red, no blue!
Sir Gawain: blue

[Knight(name='Arthur', title='King', color='blue', quest='The Grail', comment='King of
the Britons'),
 Knight(name='Galahad', title='Sir', color='red', quest='The Grail', comment="'I could
handle some more peril'"),
 Knight(name='Lancelot', title='Sir', color='blue', quest='The Grail', comment='"It\'s
too perilous!"'),
 Knight(name='Robin', title='Sir', color='yellow', quest='Not Sure', comment='He boldly
ran away'),
 Knight(name='Bedevere', title='Sir', color='red, no blue!', quest='The Grail',
comment='AARRRRRRRGGGGHH'),
 Knight(name='Gawain', title='Sir', color='blue', quest='The Grail', comment='none')]
```

Alternatives to collections.namedtuple

There are several classes or packages for auto-generating a class or class-like object:

- `typing.NamedTuple`
- `dataclasses.dataclass`
- `pydantic`[1]
- `attrs`[1]

[1] These packages must be downloaded from **PyPI**, the Python package repository.

Table 17. Comparison of tuples and data class generators

	tuple	**namedtuple**	**typing. NamedTuple**	**dataclass**	**pydantic**	**attrs**
Writable	N	N	N	Y	Y	Y
Validation	N	N	N	N[2]	Y	N[2]
Iterable	Y	Y	Y	N	N	N
Indexable	Y	Y	Y	N	N	N
Methods	N	N	Y[1]	Y	Y	Y
Type hints	N	N	Y	Y	Y	Y

[1] Only when subclassing
[2] Could be added manually

Printing data structures

- Default representation of data structures is ugly
- `pprint` makes structures human friendly
- Use `pprint.pprint()`
- Useful for debugging

When debugging data structures, the `print()` function is not so helpful. A complex data structure is just printed out all jammed together, one element after another, with ony a space between elements. This can be very hard to read.

The `pprint` (pretty print) module provides the `pprint()` function, which will analyze a structure and print it out with indenting and newlines. This makes the data easier to read.

You can customize the output with named parameters. See the following table for details.

Table 18. `pprint()` *parameters*

Parameter	Description	Default Value
`stream`	Write data to stream (stdout if None)	`None`
`indent`	How many spaces to indent each level	`1`
`width`	Only use specified number of columns	`80`
`depth`	Only display specied levels of data	`None`
`compact`	Try to display data more compactly	`False`
`sort_dicts`	Display dictionaries sorted by keys	`True`
`underscore_numbers`	Add underscores to numbers for readability	`False`

Example

pretty_printing.py

```
from pprint import pprint

struct = {  # nested data structure
    'epsilon': [
        ['a', 'b', 'c'], ['d', 'e', 'f']
    ],
    'theta': {
        'red': 55,
        'blue': [8, 98, -3],
        'purple': ['Chicago', 'New York', 'L.A.'],
    },
    'alpha': ['g', 'h', 'i', 'j', 'k'],
    'gamma': [39029384, 3827539203, 94838402, 249398063],
}

print('Without pprint (normal output:')
print(struct)  # print normally
print()

print('With pprint:')
pprint(struct)  # pretty-print
print()

print('With pprint (sort_dicts=False):')
pprint(struct, sort_dicts=False)  # Leave dictionary in default order
print()

print('With pprint (depth=2):')
pprint(struct, depth=2)  # only print top two levels of structure
print()

print('With pprint (width=40):')
pprint(struct, width=40)  # set display width
print()

print('With pprint (underscore_numbers=True):')
pprint(struct, underscore_numbers=True)  # Put underscores in large numbers for
readability
```

pretty_printing.py

```
Without pprint (normal output:
{'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']], 'theta': {'red': 55, 'blue': [8, 98, -3],
'purple': ['Chicago', 'New York', 'L.A.']}, 'alpha': ['g', 'h', 'i', 'j', 'k'], 'gamma':
[39029384, 3827539203, 94838402, 249398063]}

With pprint:
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'gamma': [39029384, 3827539203, 94838402, 249398063],
 'theta': {'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.'],
           'red': 55}}

With pprint (sort_dicts=False):
{'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'theta': {'red': 55,
           'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.']},
 'alpha': ['g', 'h', 'i', 'j', 'k'],
 'gamma': [39029384, 3827539203, 94838402, 249398063]}

With pprint (depth=2):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [[...], [...]],
 'gamma': [39029384, 3827539203, 94838402, 249398063],
 'theta': {'blue': [...], 'purple': [...], 'red': 55}}

With pprint (width=40):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'],
             ['d', 'e', 'f']],
 'gamma': [39029384,
           3827539203,
           94838402,
           249398063],
 'theta': {'blue': [8, 98, -3],
           'purple': ['Chicago',
                      'New York',
                      'L.A.'],
           'red': 55}}

With pprint (underscore_numbers=True):
{'alpha': ['g', 'h', 'i', 'j', 'k'],
 'epsilon': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'gamma': [39_029_384, 3_827_539_203, 94_838_402, 249_398_063],
 'theta': {'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.'],
           'red': 55}}
```

Zipped archives

- import `zipfile` for zipped files
- Get a list of files
- Extract files

The `zipfile` module provides the `ZipFile` class which allows you to read and write to zipped archives. In either case you first create a `ZipFile` object, specifying the name of the zip file.

Reading zip files

Create an instance of `ZipFile`, specifying the path to the zip file.

```
To get a list of members (contained files), use ((`ZIPFILE.namelist()`)).
```

```
((`ZIPFILE.getinfo("member-name")`)) will retrieve metadata about the member as a ZipInfo
object.
```

To extract a member, use `ZIPFILE.extract("member-name")`. To read the data from a member without extracting it, use `ZIPFILE.read("member-name")`. When you read member data, is is read in binary mode, so a text file will be read in as a `bytes` object. Use `.decode()` on the bytes object to get a normal Python string.

The `tarfile` module will read and write compressed or uncompressed **tar** files.

The **Zip File Explorer** extension for VS Code will display the contents of zipfiles in the Explorer panel

Example

zipfile_read.py

```
from zipfile import ZipFile

# read & extract from zip file
zip_in = ZipFile("../DATA/textfiles.zip")  # Open zip file for reading
print(zip_in.namelist())  # Print list of members in zip file
tyger_text = zip_in.read('tyger.txt').decode()  # Read (raw binary) data from member and
convert from bytes to string
print(tyger_text[:100], '\n')
zip_in.extract('parrot.txt')  # Extract member
```

zipfile_read.py

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']
           The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand o
```

Writing to zip files

To create a new zip archive, create an instance of `ZipFile`, specifying mode 'w'. Specify `ZIP_DEFLATED` (normal compression) as the compression type.

Add files to the zip archive with the `write()` method on the `ZipFile` object.

Example

zipfile_write.py

```
from zipfile import ZipFile, ZIP_DEFLATED
import os.path

file_names = ["parrot.txt", "tyger.txt", "knights.txt", "alice.txt", "poe_sonnet.txt",
"spam.txt"]
file_folder = "../DATA"

zipfile_name = "example.zip"

# creating new, empty, zip file
zip_out = ZipFile(zipfile_name, mode="w", compression=ZIP_DEFLATED)  # Create new zip
file

# add files to zip file
for file_name in file_names:
    file_path = os.path.join(file_folder, file_name)
    zip_out.write(file_path, file_name)  # Add member to zip file
zip_out.close()

# list files in zip
zip_in = ZipFile(zipfile_name)
print("Files in archive:")
print(zip_in.namelist())
```

Making archives with **shutil**

- Create archive from folder
- Simpler than `zipfile.ZipFile`

The `shutil` module makes it easy to create an archive of an entire folder. It will create archives in the following format: **zip**, **tar**, **gztar**, **bztar**.

To create an archive, call `shutil.make_archive()`. The arguments are:

1. Base name of the output file
2. Format (one of "zip", "tar", "gztar", "bztar", or "xztar").
3. Folder to be archived (current folder if omitted)

Example

shutil_make_archive.py

```
import shutil
import os

folder = '../DATA'
archive_name = "datafiles"

for archive_type in 'zip', 'gztar':
    shutil.make_archive(archive_name, archive_type, folder)
```

Serializing Data with **pickle**

- Use the `pickle` module
- Save to file
- Transmit over network

Serializing data means taking a data structure and transforming it so it can be written to a file or other destination, and later read back into the same data structure.

Python uses the `pickle` module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. `pickle.dumps()` returns the pickled data as a `bytes` object. `pickle.dump()` writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been open for writing, or `pickle.loads()` which reads from a string. Both functions return the original data structure that had been pickled.

Remember to open pickle files in binary mode.

Example

pickling.py

```
import pickle
from pprint import pprint

# some data structures
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

values = [
    3/7, 1/9, 14.5
]

data = [  # list of data structures
    colors,
    airports,
    values,
]

print("BEFORE:")
pprint(data)
print('-' * 60)



with open('../TEMP/pickled_data.pkl', 'wb') as pkl_out:  # open pickle file for writing
in binary mode
    pickle.dump(data, pkl_out)  # serialize data structures to pickle file

with open('../TEMP/pickled_data.pkl', 'rb') as pkl_in:  # open pickle file for reading in
binary mode
    pickled_data = pickle.load(pkl_in)  # de-serialize pickle file back into data
structures

print("AFTER:")
pprint(pickled_data)  # view data structures
```

pickling.py

```
BEFORE:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
------------------------------------------------------------
AFTER:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
```

Chapter 11 Exercises

Exercise 11-1 (count_ext.py)

Start with the existing script `count_ext.py`.

Add code to count the number of files with each extension in a file tree.

The script will take the starting folder as a command line argument, and then display the total number of files with each distinct file extension that it finds. Files with no extension should be skipped. Use a `Counter` object to do the counting.

Use `os.path.splitext()` to split the filename into a **path**,**extension** tuple.

Exercise 11-2 (prestuple.py, save_potus_info.py, read_potus_info.py)

Part A

Create a module named `prestuple` which defines a named tuple `President`, with fields **term**, **lastname**, **firstname**, **birthstate**, and **party**.

Part B

Write a script that uses the `csv` module to read the data from `presidents.csv` into a list of `President` named tuples. The script can import the `President` named tuple from the `prestuple` module.

Use the `pickle` module to write the list out to a file named **potus.pkl**.

Part C

Write a script to open `potus.pkl`, and restore the data back into an array.

Then loop through the array and print out each president's first name, last name, and party.

The `prestuple` module is not needed for this script.

Exercise 11-3 (make_zip.py)

Write a script which creates a zip file containing `save_potus_info.py`, `read_potus_info.py`, and `potus.pkl`.

Use `zipfile.ZipFile`

Chapter 12: Iterables and Generators

Objectives

- Understand iterables vs iterators
- Develop iterator classes
- Create iterators with generators
- Unpack iterables with wildcards
- Unpack function arguments

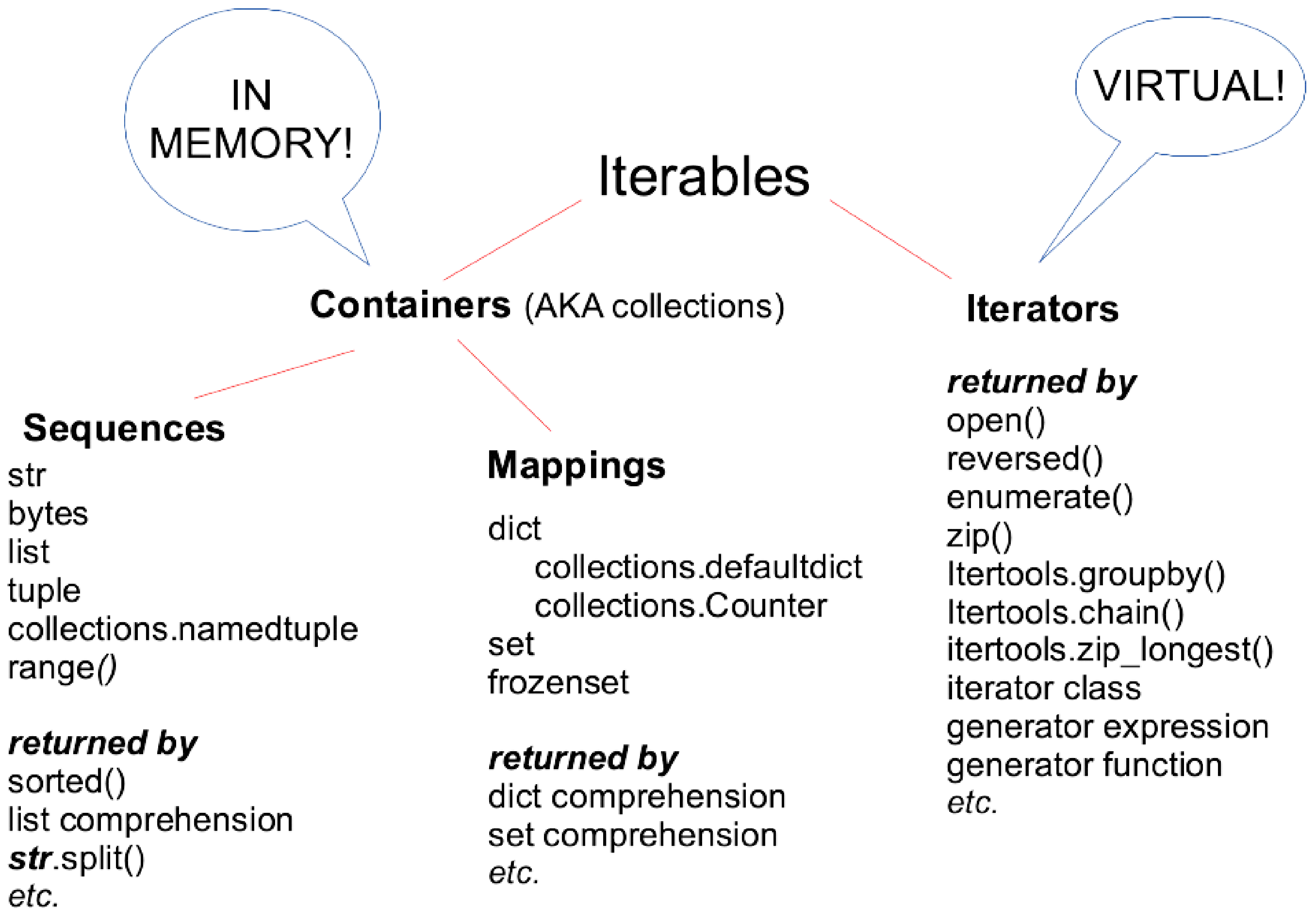
Iterables

- An iterable is an expression that can be looped through with `for`
- Some iterables are collections (`list`, `tuple`, `str`, `bytes`, `dict`, `set`)
- Many iterables are *iterators* (`enumerate()`, `dict.items()`, `open()`, `zip()`, `reversed()`, etc)

`for` is a very useful Python operator. It it used for looping through a sequence of values. The sequence of values is provided by an *iterable*. Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All collections (`list`, `tuple`, `str`, `bytes`, etc.) are iterables. They keep all their values in memory.

An *iterator* is an iterable that does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for loop. This saves memory.



What *exactly* is an iterable?

- Object that provides an *iterator*
- Can be **collection**, **generator**, or **iterator**

An *iterable* is an object that provides an *iterator* (via the special method `__iter__()`. An *iterator* is an object that responds to the `next()` builtin function, via the special method `__next__()`.

Any iterable can be looped over with a `for` loop.

All iterators and generators are iterables. Sequence and mapping types are also iterables.

For some iterables (including collections), you can't use `next()` on them directly; first, use the builtin function `iter()` to get the iterator.

A *generator* is an iterator created with a generator expression or a generator function. These are convenient shortcuts to creating an iterator object. You can also define a class which provides an iterator.

```
>>> r = range(1, 4)
>>> it = iter(r)   # range is not an iterator
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Iterators

- Lazy iterables
- Do not contain data
- Provide values on demand
- Save memory

An *iterator* is an object that provides values on demand (AKA "lazy"), rather than storing all the values (AKA "eager"). An iterator returns a new value each time `next()` is called on it, until there are no more values.

The advantage of iterators is saving memory. They can act as a *view* over a set of data.

Most iterators may only be used once. After the last value is provided, a new iterator must be created in order to start over.

Iterators are read-only, may not be indexed, and have no length. The only operations possible on an iterator is to loop over it with `for`, or consume the next element with `next()`.

Python has many builtin iterators, and there are three ways to create user-defined iterators in Python:

- iterator class
- generator expression
- generator function

Think of an iterator as a "value factory".

A `for` loop is really a `while` loop in disguise. It repeatedly calls `next()` on an iterator, and stops when `StopIteration` is raised.

Table 19. Iterables

Type	**is it an iterable?** *implements `__iter__()`*	**Is it an iterator?** *implements `__next__()`*
builtin iterator (file object, `enumerate()`, `reversed()`, etc.)	yes	yes
iterator class instance	yes	probably
generator expression generator function	yes	yes
collection (`list`, `dict`, `set`, `tuple`, etc)	yes	no

How a `for` loop really works

```
values = ["a", "b", "c"]
```

`for` loop

```
for value in values:
   print(value)
```

equivalent `while` loop

```
it = iter(values)
while True:
   try:
      value = next(it)
   except StopIteration:
      break
   print(value)
```

Iterator classes

- Implement `__iter__()` and `__next__()`
- Emit values via **for** or **next()**
- Raise StopIteration exception when there are no more values to emit

The most flexible way to create an iterator is by defining an iterator class.

Such a class *must* implement `__iter__()`, and usually also implements `__next__()`:

`__iter__()` must return an iterator — an object that implements `__next__()`. In most cases, this is the same class, so `__iter__()` just returns `self`.

`__next__()` returns the next value from the iterator.

When there are no more values to return, `__next__()` should raise `StopIteration`.

Example

trimmedfile.py

```
class TrimmedFile:
    def __init__(self, file_name):  # constructor is passed file name
        self._file_in = open(file_name)

    def __iter__(self):
        # An iterator must implement iter(), which must return an iterator.
        # Typically it returns 'self', as the generator IS the iterator
        return self

    def __next__(self):  # next() returns the next value of the iterator
        line = self._file_in.readline()
        if line == '':
            raise StopIteration  # Raise StopIteration when there are no more values to
generate
        else:
            return line.rstrip('\n\r')  # The actual work of this iterator -- remove the
end-of-line char(s)


if __name__ == '__main__':
    # To use the iterator, create an instance and iterate over it.
    trimmed = TrimmedFile('../DATA/mary.txt')
    for line in trimmed:
        print(line)
```

trimmedfile.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

Generator Expressions

- Like list comprehensions, but create a generator object
- Use parentheses rather than brackets
- Saves memory

A generator expression is similar to a list comprehension, but it provides an iterator instead of a list. That is, while a list comprehension returns a complete list, the generator expression an iterator. The iterator does not contain a copy of the source iterable (it has a *reference* to it), so it saves memory. In many cases generator expressions are also faster than list comprehensions.

The difference in syntax is that a generator expression is surrounded by parentheses, while a list comprehension uses brackets.

```
colors = ['blue', 'red', 'purple']
upper_colors = [c.upper() for c in colors]  # list comprehension
upper_colors_gen = (c.upper() for c in colors)  # generator expression
```

If a generator expression is passed as the only argument to a function, the surrounding parentheses are not needed:

```
my_func(float(i) for i in values)
```

Example

generator_expressions.py

```
fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

ufruits = (fruit.upper() for fruit in fruits)  # These are all exactly like the list
comprehension example, but return generators rather than lists
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a'))

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print()

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]
doubles = (v * 2 for v in values)

print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print("\n")

nums = (int(s) for s in values if isinstance(s, int))
for n in nums:
    print(n, end=' ')
print("\n")

dirty_strings = ['   Gronk    ', 'PULABA       ', '        floog']

clean = (d.strip().lower() for d in dirty_strings)
for c in clean:
    print(f">{c}<", end=' ')
print("\n")

powers = ((i, i ** 2, i ** 3) for i in range(1, 11))
for num, square, cube in powers:
    print(f"{num:2d} {square:3d} {cube:4d}")
```

generator_expressions.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: Apple Apricot

doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']

2 42 18 92

>gronk< >pulaba< >floog<

 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Generator functions

- Mostly like a normal function
- Use `yield` rather than `return`
- Maintains state

A generator function is defined like a normal function, but instead of a return statement, it has a `yield` statement.

The value returned by the generator function is the generator (iterator) itself.

The code in the function does not execute until the generator is passed to `next()` or is used with `for`. When the `for` loop starts, the code in the function executes up to the point of the `yield` statement. The code is paused, and the value of the `yield` statement is copied to the loop variable. On the next iteration of the loop, the function code continues running until it gets to `yield` again.

Each time the `yield` statement is reached, it provides the next value in the sequence. When there are no more values, the function returns. A generator function maintains state between calls, unlike a normal function.

You can think of a generator function as a "generator factory".

Example

sieve_generator.py

```
def next_prime(limit):
    flags = set()  # initialize empty set (to be used for "is-prime" flags

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  # add non-prime elements to set
        yield i  # execution stops here until next value is requested by for-in loop


np = next_prime(200)  # next_prime() returns a generator object
for prime in np:  # iterate over yielded primes
    print(prime, end=' ')
```

sieve_generator.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

line_trimmer.py

```
def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            yield line.rstrip('\n\r')  # 'yield' causes this function to return a
generator object

mary_in = trimmed('../DATA/mary.txt')
for trimmed_line in mary_in:
    print(trimmed_line)
```

line_trimmer.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

Iterable unpacking

- Frequently used with `for` loops
- Can be used anywhere
- Commonly used with `enumerate()` and `DICT.items()`
- Extending unpacking uses `*`

Basic unpacking

It is convenient to unpack an iterable into a list of variables. It can be done with any iterable, and is frequently done with the loop variables of a `for` loop, rather than unpacking the value of each iteration separately.

```
var1, … = iterable
```

Extended unpacking

When unpacking iterables, sometimes you want to grab parts of the iterable as a group. This is provided by extended iterable unpacking. This can also be useful for unpacking nested iterables, where each element of a sequence is an iterable, but the nested iterables are not all the same length.

One (and only one) variable in the result of unpacking can have a star prepended. This variable will receive all values from the iterable that do not go to other variables.

```
a, *b, c = 1, 2, 3, 4, 5, 6
```

Underscore (`_`) can be used as a variable name for values you don't care about.

Example

iterable_unpacking.py

```
values = ['a', 'b', 'c']

x, y, z = values  # unpack values (which is an iterable) into individual variables

print(x, y, z)
print()

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for row in people:
    first_name, last_name, _ = row  # unpack row into variables
    print(first_name, last_name)
print()

for first_name, last_name, _ in people:  # a for loop unpacks if there is more than one
variable
    print(first_name, last_name)
print()
```

iterable_unpacking.py

```
a b c

Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds

Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Example

extended_iterable_unpacking.py

```
values = ['a', 'b', 'c', 'd', 'e']  # values has 6 elements

x, y, *z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

x, *y, z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

*x, y, z = values  # * takes all extra elements from iterable
print(f"x: {x}    y: {y}    z: {z}\n")

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for *name, _ in people:  # name gets all but the last field
    print(name)
print()
```

extended_iterable_unpacking.py

```
x: a    y: b    z: ['c', 'd', 'e']

x: a    y: ['b', 'c', 'd']    z: e

x: ['a', 'b', 'c']    y: d    z: e

['Bill', 'Gates']
['Steve', 'Jobs']
['Paul', 'Allen']
['Larry', 'Ellison']
['Mark', 'Zuckerberg']
['Sergey', 'Brin']
['Larry', 'Page']
['Linux', 'Torvalds']
```

Unpacking function arguments

- Convert from iterable to list of items
- Use * or **

What do you do if you have a list (or other iterable) of three values, and you want to pass them to a method that expects three positional arguments?

You could say value[0], value[1], value[2], etc., but there's a more Pythonic way:

Use * to unpack any iterable into individual items. You can combine individual arguments with unpacking:

```
values = [1, 2, 3]
names = ['Rodrigo', 'Srini']
colors = ['pink', 'orange']
foo(5, 10, *values)   # same as foo(5, 10, 1, 2, 3)
bar('blue', *names, 55, *colors)  # same as bar('blue', 'Rodrigo', 'Srini', 55, 'pink',
'orange')
```

In a similar way, use two asterisks to unpack a dictionary.

```
config_opts = {'filename': 'foo.txt', 'folder': 'ProjectAlpha'}
config(**config_opts)  # same as config(filename='foo.txt', folder='ProjectAlpha')
```

Example

argument_unpacking.py

```
from datetime import date

dates = [
    (1968, 10, 11),
    (1968, 12, 21),
    (1969, 3, 3),
    (1969, 5, 18),
    (1969, 7, 16),
    (1969, 11, 14),
    (1970, 4, 11),
    (1971, 1, 31),
    (1971, 7, 26),
    (1972, 4, 16),
    (1927, 12, 7),
]  # tuple of dates

for dt in dates:
    d = date(*dt)  # instead of date(dt[0], dt[1], dt[2])
    print(d)

print()

fruits = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
          "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear",
          "banana", "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry",
          "lychee", "grape"]

sort_opts = {
    'key': lambda e: e.lower(),
    'reverse': True,
}  # config info in dictionary

sorted_fruits = sorted(fruits, **sort_opts)  # dictionary converted to named parameters
print(sorted_fruits)
```

argument_unpacking.py

```
1968-10-11
1968-12-21
1969-03-03
1969-05-18
1969-07-16
1969-11-14
1970-04-11
1971-01-31
1971-07-26
1972-04-16
1927-12-07

['Watermelon', 'Tamarind', 'pomegranate', 'persimmon', 'pear', 'peach', 'papaya',
'ORANGE', 'lychee', 'lime', 'lemon', 'Kiwi', 'guava', 'grape', 'FIG', 'elderberry',
'date', 'cherry', 'BLUEberry', 'banana', 'apricot', 'Apple']
```

Chapter 12 Exercises

Exercise 12-1 (pres_upper_gen.py)

Step 1

Create a generator expression to read all the presidents' first and last names into a sequence of tuples.

Step 2

Create a second generator expression that uses the first one to make a new sequence with the names joined by spaces, and in upper case.

Step 3

Loop through the second generator and print out the names one per line.

Exercise 12-2 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file. Write code to loop over the generator and print out the names.

Exercise 12-3 (fauxrange.py)

Write a generator class named `FauxRange` that emulates the builtin `range()` function. Instances should take up to three arguments, and provide a range of integers.

Chapter 13: Functional Tools

Objectives

- Conceptualize higher-order functions
- Learn what’s in functools and itertools
- Transform data with map/reduce
- Chain multiple iterables together
- Implement function overloading with single dispatch
- Use iterative tools to work with iterators
- Chain multiple iterators together
- Compute combinations and permutations
- Zip multiple iterables with default values

Higher-order functions

- Functions that operate on (or return) functions

Higher-order functions operate on or return functions. The most traditional higher-order functions in other languages are `map()` and `reduce()`. These are used for functional programming, which prevents side effects (modifying data outside the function).

Some languages make it difficult or impossible to pass functions into other functions, but in Python, since functions are first-class objects, it is easy.

Higher-order functions may be nested, to have multiple transformations on data.

Lambda functions

- Inline anonymous functions
- Contain only parameters and return value
- Useful as predicates in higher-order functions

A **lambda** function is an inline anonymous function declaration. It evaluates as a function object, in the same way that `def function(…):` … does.

A lambda declaration has only parameters and the return value. Blocks are not allowed, nor is the **return** keyword.

Lambdas are useful as predicates (callbacks) in higher-order functions.

Example

higher_order_functions.py

```
fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

def process_list(alist, func):  # Define a function that accepts a list and a passed-in
function (AKA callback)
    new_list = []
    for item in alist:
        new_list.append(func(item))  # Call the callback function on one item of the
passed-in list
    return new_list

f1 = process_list(fruits, str.upper)  # Call process_list() with str.upper as the
callback
print(f"f1: {f1}\n")

f2 = process_list(fruits, lambda s: s[0].upper())  # Call process_list() with a lambda
function as the callback
print(f"f2: {f2}\n")

f3 = process_list(fruits, len)  # Call process_list() with the builtin function len() as
the callback()
print(f"f3: {f3}\n")

total_length = sum(process_list(fruits, len))  # Pass the result of process_list() to the
builtin function sum() to sum all the values in the returned list
print(f"total_length: {total_length}")
```

higher_order_functions.py

```
f1: ['POMEGRANATE', 'CHERRY', 'APRICOT', 'DATE', 'APPLE', 'LEMON', 'KIWI', 'ORANGE',
'LIME', 'WATERMELON', 'GUAVA', 'PAPAYA', 'FIG', 'PEAR', 'BANANA', 'TAMARIND',
'PERSIMMON', 'ELDERBERRY', 'PEACH', 'BLUEBERRY', 'LYCHEE', 'GRAPE']

f2: ['P', 'C', 'A', 'D', 'A', 'L', 'K', 'O', 'L', 'W', 'G', 'P', 'F', 'P', 'B', 'T', 'P',
'E', 'P', 'B', 'L', 'G']

f3: [11, 6, 7, 4, 5, 5, 4, 6, 4, 10, 5, 6, 3, 4, 6, 8, 9, 10, 5, 9, 6, 5]

total_length: 138
```

The operator module

- Provides operators as functions
- Use rather than simple lambdas

The `operator` module provides functional versions of Python's standard operators. This saves the trouble of creating trivial lambda functions.

Instead of

```
lambda x, y: x + y
```

You can just use

```
operator.add
```

Both of these functions take two operators and add them, return the result.

Example

operator_module.py

```
from operator import add



a = 10
b = 15

print(f"a + b: {a + b}")  # Add with add operator
print(f"add(a, b): {add(a, b)}")  # Add with add function
```

operator_module.py

```
a + b: 25
add(a, b): 25
```

The functools module

- Included in standard library
- Supports higher-order programming
- Many standard higher-order functions

The `functools` module provides tools for higher-ordering programming, which means functions that operate on, or return functions (some do both). Functional programming avoids explicit loops.

`map()` and `reduce()` are two functions that are the basis of many functional algorithms. `map()` is a builtin function. `reduce()` *was* builtin in Python 2, but in Python 3 must be imported from `functools`.

Most of these functional algorithms can also be implemented with list comprehensions or generator expressions, which many people find easier to both read and write.

map()

- Applies function to every element of iterable
- Syntax
  - `map(function, iterable)`

map() returns a virtual list (i.e., a generator) created by applying a function to every element of an iterable.

`map(func, list)` returns an iterator like `[func(list[0]), func(list[1]), func(list[2], …)]`

The first argument to `map()` is a function that takes one argument. Each element of the iterable is passed to the function, and the return value is added to the result generator.

Example

using_map.py

```
#

strings = ['wombat', 'koala', 'kookaburra', 'blue-ringed octopus']

result = [s.upper() for s in strings]  # Using a list comprehension, which is usually
simpler than map()
print(result)

result = list(map(str.upper, strings))  # Using map to copy list to upper case
print(result)

result = list(map(len, strings))  # Using map to get list of string lengths
print(result)
```

using_map.py

```
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
[6, 5, 10, 19]
```

reduce()

- Imported from `functools`
- Applies function to every element plus previous result

`reduce()` returns the value created by applying a function to every element of a list *and* the previous function result. `reduce()` must be imported from the `functools` module.

`reduce(func, list)` returns `func(list[n], func(list[n-1], func(list[2]…func(list[0]))))`

A third argument to `reduce()` provides the initial value. By default, the initial value is the first element of the list.

Other functions such as `sum()` or `str.join()` can be defined in terms of `map()` or `reduce()`.

The **mapreduce** approach to massively parallel processing, such as Hadoop, was inspired by map() and reduce().

Example

using_reduce.py

```
#
from operator import add, mul
from functools import reduce

values = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# sum()
result = reduce(add, values) # add values in list (initial value defaults to 0)
print("result is", result)

# sum() + 1000
result = reduce(add, values, 1000)  # add values in list (initial value is 1000)
print("result is", result)

# product
result = reduce(mul, values)  # multiply all values together (initial value is 1,
otherwise product would be 0)
print("result is", result)

strings = ['fi', 'fi', 'fo', 'fum']

# join
result = reduce(add, strings, "") # concatenate strings (initial value is empty string;
each string in iterable added to it)
print("result is", result)

# join + upper case
result = reduce(add, map(str.upper, strings), "")  # same, but make strings upper case
print("result is", result)
```

using_reduce.py

```
result is 550
result is 1550
result is 3628800000000000
result is fififofum
result is FIFIFOFUM
```

Partial functions

- Some arguments filled in
- Create desired signature

Partial functions are wrappers that have some arguments already filled in for the "real" function. This is especially useful when creating callback functions. It is also nice for creating customized functions that rely on functions from the standard library.

While you can create partial functions by hand, using closures, the `partial()` function (from the `functools` module) simplifies creating such a function.

The arguments to `partial()` are the function and one or more arguments. It returns a new function object, which will call the specified function and pass in the provided argument(s).

Example

partial_functions.py

```
#
import re

from functools import partial

count_by = partial(range, 0, 25)  # create partial function that "preloads" range() with
arguments 0 and 25

print((list(count_by(1))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(3))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(5))))  # call partial function with parameter, 0 and 25
automatically passed in
print()

has_a_number = partial(re.search, r'\d+')  # create partial function that embeds pattern
in re.search()

strings = [
    'abc', '123', 'abc123', 'turn it up to 11', 'blah blah'
]

for s in strings:
    print(f"{s}:", end=' ')
    if has_a_number(s): # call re.search() with specified pattern
        print("YES")
    else:
        print("NO")
```

partial_functions.py

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24]
[0, 3, 6, 9, 12, 15, 18, 21, 24]
[0, 5, 10, 15, 20]

abc: NO
123: YES
abc123: YES
turn it up to 11: YES
blah blah: NO
```

Single dispatch

- Provides generic functions
- Emulates function/method overloading
- Functions registered by signature

The `singledispatch` class provides *generic functions*. A generic function is defined once with the `singledispatch()` decorator, then other functions may be registered to it.

To register a function, decorate it with `@original_function.register(type)`. Then, when the original function is passed an argument of type type as its first parameter, it will call the registered function instead. If no registered functions are found for the type, it will call the original function.

This can of course, be done manually by checking argument types, then calling other methods, but using `singledispatch` is much less cumbersome.

To check which function would be chosen for a given type, use the `dispatch()` method of the original function. To get a list of all registered functions, use the registry attribute.

The `singledispatch` module was added to the standard library beginning with Python 3.4.

Because it only works for the first parameter, `singledispatch` was not in the past useful for class methods, but starting with Python 3.8, the `functools.singledispatchmethod` class provides single dispatch for methods.

Example

single_dispatch.py

```
from functools import singledispatch

@singledispatch
def add_number(number, list_object):  # define function to be called
    """Add a number to a list, converting from bool, str, bytes,
    or None if needed

    :param number: number to add
    :type number: Union[int, float, bool, str, bytes, None]
    :param list_object: target of append operation
    :type list_object: list
    """
    source_type = type(number).__name__
    raise TypeError(f"Invalid arg: must be int, float, bool, str, or bytes, not
{source_type}")


@add_number.register(int)
@add_number.register(float)
def _(number, list_object): # define handler for numeric types
    list_object.append(number)

@add_number.register(bool)
def _(number, list_object):
    list_object.append(int(number))

@add_number.register(type(None))
def _(number, list_object):
    list_object.append(0)

@add_number.register(str)
@add_number.register(bytes)
def _(number, list_object): # define handler for str and bytes
    if _has_dot(number):
        number_class = float
    else:
        number_class = int
    try:
        f = number_class(number)
    except Exception:
        raise TypeError("Cannot convert '{number}'")
    else:
        list_object.append(f)
```

```
def _has_dot(n):
    if isinstance(n, str) and '.' in n:
        return True
    elif isinstance(n, bytes) and b'.' in n:
        return True
    else:
        return False


if __name__ == "__main__":
    print('-' * 60)
    x = [1, 2, 3]
    add_number(10, x)
    add_number(20.0, x)
    add_number(True, x)
    add_number("30", x)
    add_number(b'40', x)
    add_number(None, x)
    print(f"x: {x}")

    print(add_number.dispatch(str))  # show handler function for str
    print()

    for arg_type, func in add_number.registry.items():
        # show functions for each registered type
        print(f"{arg_type!s:20} {func}")

    print(f"sum(x): {sum(x)}")  # every element is a number
```

single_dispatch.py

```
------------------------------------------------------------
x: [1, 2, 3, 10, 20.0, 1, 30, 40, 0]
<function _ at 0x10ed65620>

<class 'object'>    <function add_number at 0x10eb9c860>
<class 'float'>     <function _ at 0x10ed64900>
<class 'int'>       <function _ at 0x10ed64900>
<class 'bool'>      <function _ at 0x10ed64860>
<class 'NoneType'>  <function _ at 0x10ed649a0>
<class 'bytes'>     <function _ at 0x10ed65620>
<class 'str'>       <function _ at 0x10ed65620>
sum(x): 107.0
```

The itertools module

- Tools to help with iteration
- Provide many types of iterators

The `itertools` module provides many different iterators. Some of the tools work on existing iterators, while others create them from scratch.

These tools work well with functions from the operator module, as well as interacting with the functional tools described earlier.

Many of the constructors in `itertools` were inspired by Haskell, SML, and APL.

See **iterable_recipes.py** in the EXAMPLES folder for a group of utility functions that use many of the routines in this chapter.

Infinite iterators

- Iterate infinitely, or specified number of times
- Cycle over or repeat values

Infinite iterators return iterators that will iterate a specified number of times, or infinitely. These iterators are similar to generators; they do not keep all the values in memory.

`islice()` selects a slice of an iterator. You can specify an iterable and up to 3 slice arguments – start, stop, and increment, similar to the the slice arguments of a list. If just stop is provided, it stops the iterator after than many values.

`count()` is similar to `range()`. It provides a sequence of numbers with a specified increment. The big difference is that there is no end condition, so it will increment forever. You will need to to test the values and stop, or use islice().

`cycle()` loops over an iterable repeatedly, going back to the beginning each time the end is reached.

`repeat()` repeats a value infinitely, or a specified number of times.

Example

infinite_iterators.py

```
#
from itertools import islice, count, cycle, repeat

for i in count(0, 10):  # count by tens starting at 0 forever
    if i > 50:
        break  # without a check, will never stop
    print(i, end=' ')
print("\n")

for i in islice(count(0, 10), 6):  # saner, using islice to get just the first 6 results
    print(i, end=' ')
print("\n")

giant = ['fee', 'fi', 'fo', 'fum']

for i in islice(cycle(giant), 10):  # cycle over values in list forever (use islice to
stop)
    print(i, end=' ')
print("\n")

for i in repeat('tick', 10):  # repeat value 10 times (default is repeat forever)
    print(i, end=' ')
print("\n")
```

infinite_iterators.py

```
0 10 20 30 40 50

0 10 20 30 40 50

fee fi fo fum fee fi fo fum fee fi

tick tick tick tick tick tick tick tick tick tick
```

Extended iteration

- Extended iteration over multiple objects
- Truncated iteration over a list

Another group of iterator functions provides extended iteration.

`chain()` takes two or more iterables, and treats them as a single iterable.

To chain the elements of a single iterable together, use `chain.from_iterable()`.

`dropwhile()` skips leading elements of an iterable until some condition is reached. `takewhile()` stops iterating when some condition is reached.

Example

extended_iteration.py

```
#
from itertools import chain, takewhile, dropwhile

spam = ['alpha', 'beta', 'gamma']
ham = ['delta', 'epsilon', 'zeta']

for letter in chain(spam, ham):  # treat spam and ham as a single iterable
    print(letter, end=' ')
print("\n")

eggs = [spam, ham]

for letter in chain.from_iterable(eggs):  # treat all elements of eggs as a single
iterable
    print(letter, end=' ')
print("\n")

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

for fruit in takewhile(lambda f: len(f) > 4, fruits):  # iterate over elements of fruits
as long as length of current item > 4
    print(fruit, end=' ')
print("\n")

for fruit in takewhile(lambda f: f[0] != 'k', fruits):  # iterate over elements of
fruits as long as fruit does not start with 'k'
    print(fruit, end=' ')
print("\n")

values = [5, 18, 22, 31, 44, 57, 59, 61, 66, 70, 72, 78, 90, 99]

for value in dropwhile(lambda f: f < 50, values):  # skip over elements of values as long
as value is < 50, then iterate over all remaining elements
    print(value, end=' ')
print("\n")
```

extended_iteration.py

```
alpha beta gamma delta epsilon zeta

alpha beta gamma delta epsilon zeta

pomegranate cherry apricot

pomegranate cherry apricot date apple lemon

57 59 61 66 70 72 78 90 99
```

Grouping

- Groups consecutive elements by value
- Input must be sorted

The `groupby()` function groups consecutive elements of an iterable by value. As with sorted(), the value my be determined by a key function. This is similar to sort -u or the uniq commands in Linux.

groupby() returns an iterable of subgroups, which can then be converted to a list or iterated over. Each subgroup has a key, which is the common value, and an iterable of the values for that key.

For a more real-life example of grouping, see `group_dates_by_week.py` in the EXAMPLES folder.

Example

groupby_examples.py

```
from itertools import groupby

with open('../DATA/words.txt') as words_in:  # open file for reading
    # create generator of all words, stripped of the trailing '
    all_words = (w.rstrip() for w in words_in)

    # create a groupby() object where the key is the first character in the word
    g = groupby(all_words, key=lambda e: e[0])

    # make a dictionary where the key is the first character, and the value
    # is the number of words that start with that character; groupby groups
    # all the words, then len() counts the number of words for that character
    counts = {letter: len(list(wlist)) for letter, wlist in g}

# sort the counts dictionary by value (i.e., number of words, not the letter
#  itself) into a list of tuples
sorted_letters = sorted(counts.items(), key=lambda e: e[1], reverse=True)

# loop over the list of tuples and print the letter and its count
for letter, count in sorted_letters:
    print(letter, count)
print()

# sum all the individual counts and print the result
print("Total words counted:", sum(counts.values()))
```

groupby_examples.py

```
s 19030
c 16277
p 14600
a 10485
r 10199
d 10189
m 9689
b 9325
t 8782
e 7086
i 6994
f 6798
h 6333
o 5858
g 5599
u 5095
l 5073
n 4411
w 3727
v 2741
k 1734
j 1378
q 827
y 552
z 543
x 137

Total words counted: 173462
```

Combinatoric generators

- Provides products, combinations and permutation
- Can specify max length of sub-sequences

Several functions provide products, combinations, and permutations.

product() return an iterator with the Cartesian product of two iterables.

combinations() returns the unique n-length combinations of an iterator.

permutations() returns all n-length sub-sequences of an iterator.

If the length is not specified, all sub-sequences are returned.

Example

combinations_permutations.py

```
#
from itertools import product, permutations, combinations

SUITS = 'CDHS'
RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

cards = product(SUITS, RANKS)  # Cartesian product (match every item in one list to every
item in the other list)
print(list(cards), '\n')

cards = [r + s for r, s in product(SUITS, RANKS)]  # reverse order and concatenate
elements using list comprehension
print(cards, '\n')

giant = ['fee', 'fi', 'fo', 'fum']

result = combinations(giant, 2)  # all distinct combinations of 4 items taken 2 at a time
print(list(result), "\n")

result = permutations(giant, 2)  # all distinct permutations of 4 items taken 2 at a time
print(list(result), "\n")
```

combinations_permutations.py

```
[('C', '2'), ('C', '3'), ('C', '4'), ('C', '5'), ('C', '6'), ('C', '7'), ('C', '8'),
('C', '9'), ('C', '10'), ('C', 'J'), ('C', 'Q'), ('C', 'K'), ('C', 'A'), ('D', '2'),
('D', '3'), ('D', '4'), ('D', '5'), ('D', '6'), ('D', '7'), ('D', '8'), ('D', '9'), ('D',
'10'), ('D', 'J'), ('D', 'Q'), ('D', 'K'), ('D', 'A'), ('H', '2'), ('H', '3'), ('H',
'4'), ('H', '5'), ('H', '6'), ('H', '7'), ('H', '8'), ('H', '9'), ('H', '10'), ('H',
'J'), ('H', 'Q'), ('H', 'K'), ('H', 'A'), ('S', '2'), ('S', '3'), ('S', '4'), ('S', '5'),
('S', '6'), ('S', '7'), ('S', '8'), ('S', '9'), ('S', '10'), ('S', 'J'), ('S', 'Q'),
('S', 'K'), ('S', 'A')]

['C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10', 'CJ', 'CQ', 'CK', 'CA', 'D2',
'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10', 'DJ', 'DQ', 'DK', 'DA', 'H2', 'H3',
'H4', 'H5', 'H6', 'H7', 'H8', 'H9', 'H10', 'HJ', 'HQ', 'HK', 'HA', 'S2', 'S3', 'S4',
'S5', 'S6', 'S7', 'S8', 'S9', 'S10', 'SJ', 'SQ', 'SK', 'SA']

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fo'), ('fi', 'fum'), ('fo',
'fum')]

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fee'), ('fi', 'fo'), ('fi',
'fum'), ('fo', 'fee'), ('fo', 'fi'), ('fo', 'fum'), ('fum', 'fee'), ('fum', 'fi'),
('fum', 'fo')]
```

Chapter 13 Exercises

Exercise 13-1 (sum_of_values.py)

Read in the data from float_values.txt and print out the sum of all values. Do this with functional tools – there should be no explicit loops in your code.

use reduce() + operator.add on the file object.

Exercise 13-2 (pres_by_state_func.py)

Using presidents.txt, print out a list of the number of presidents from each state.

Use map() + lambda to split lines from presidents.txt on the 7th field, then use groupby() on that.

Exercise 13-3 (count_all_lines.py)

Count all of the lines in all the files specified on the command line without using any loops.

use map() + chain.from_iterable() to create an iterable of all the lines, then use reduce to count them.

Chapter 14: Introduction to NumPy

Objectives

- See the "big picture" of NumPy
- Create and manipulate arrays
- Learn different ways to initialize arrays
- Understand the NumPy data types available
- Work with shapes, dimensions, and ranks
- Broadcast changes across multiple array dimensions
- Extract multidimensional slices
- Perform matrix operations

Python’s scientific stack

- NumPy, SciPy, MatPlotLib (and many others)
- Python extensions, written in C/Fortran
- Support for math, numerical, and scientific operations

NumPy is part of what is sometimes called Python’s "scientific stack". Along with SciPy, Matplotlib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

SciPy is a large group of mathematical algorithms, along with some convenience functions, for doing scientific and engineering calculations, including data science. SciPy routines accept and return NumPy arrays.

pandas ties some of the libraries together, and is frequently used interactively via **iPython** in a **Jupyter** notebook. Of course you can also create scripts using any of the scientific libraries.

See https://www.numpy.org for details. At the bottom of the home page there is a good summary of the Python ecosystem for scientific computing and data analysis.

There is not an integrated *application* for all of the Python scientific libraries.

NumPy overview

- Install numpy module from numpy.scipy.org (included with Anaconda)
- Basic object is the array
- Up to 100x faster than normal Python math operations
- Functional-based (fewer loops)
- Dozens of utility functions

The basic object that NumPy provides is the array. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the **numpy** module. It is conventional to import numpy as **np**. The examples in this chapter will follow that convention.

NumPy and the rest of the Python scientific stack is included with the Anaconda, Canopy, Python(x,y), and WinPython bundles. If you are not using one of these, install NumPy with

```
pip install numpy
```

all top-level NumPy routines are also available directly through the scipy package.

Creating Arrays

- Create with
  - array() function initialized with nested sequences
  - Other utilities (arange(), zeros(), ones(), empty()
- All elements are same type (default float)
- Useful properties: ndim, shape, size, dtype
- Can have any number of axes (dimensions)
- Each axis has a length

An array is the most basic object in NumPy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the `array()` function, which can be initialized from existing arrays.

The `zeros()` function expects a *shape* (tuple of axis lengths), and creates the corresponding array, with all values set to zero. The `ones()` function is the same, but initializes with ones.

The `full()` function expects a shape and a value. It creates the array, putting the specified value in every element.

The `empty()` function creates an array of specified shape initialized with random floats.

However, the most common way to crate an array is by loading data from a text or binary file.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

the `ndarray()` object is initialized with the *shape*, not the *data*.

Example

np_create_arrays.py

```
import sys
import numpy as np
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [20, 30, 40]]

a = np.array(data)  # create array from nested sequences
print(a, '\n')

print("a.ndim (# dimensions):", a.ndim)  # get number of dimensions
print("a.shape (lengths of axes/dimensions):", a.shape)  # get shape
print("a.size (number of elements in array):", a.size)
print("a.itemsize (size of one item):", a.itemsize)
print("a.nbytes (number of bytes used):", a.nbytes)
print("sys.getsizeof(data):", sys.getsizeof(data))
print()

a_zeros = np.zeros((3, 5), dtype=np.uint32)  # create array of specified shape and
datatype, initialized to zeroes
print(a_zeros)
print()

a_ones = np.ones((2, 3, 4, 5))  # create array of specified shape, initialized to ones
print(a_ones)
print()

# with uninitialized values
a_empty = np.empty((3, 8))  # create uninitialized array of specified shape
print(a_empty)

print(a.dtype)  # defaults to float64

nan_array = np.full((5, 10), np.NaN)  # create array of NaN values
print(nan_array)
```

np_create_arrays.py

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [20 30 40]]

a.ndim (# dimensions): 2
a.shape (lengths of axes/dimensions): (4, 3)
a.size (number of elements in array): 12
a.itemsize (size of one item): 8
a.nbytes (number of bytes used): 96
sys.getsizeof(data): 88

[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

[[[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]


 [[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]]

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
int64
[[nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]]
```

Creating ranges

- Similar to builtin `range()`
- Returns a one-dimensional NumPy array
- Can use floating point values
- Can be reshaped

The `arange()` function takes a size, and returns a one-dimensional NumPy array. This array can then be reshaped as needed. The start, stop, and step parameters are similar to those of `range()`, or Python slices in general. Unlike the builtin Python `range()`, start, stop, and step can be floats.

The `linspace()` function creates a specified number of equally-spaced values. As with `numpy.arange()`, start and stop may be floats.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
import numpy as np

r1 = np.arange(50)  # create range of ints from 0 to 49
print(r1)
print("size is", r1.size)  # size is 50
print()

r2 = np.arange(5, 101, 5)  # create range of ints from 5 to 100 counting by 5
print(r2)
print("size is", r2.size)
print()

r3 = np.arange(1.0, 5.0, .3333333)  # start, stop, and step may be floats
print(r3)
print("size is", r3.size)
print()

r4 = np.linspace(1.0, 2.0, 10)  # 10 equal steps between 1.0 and 2.0
print(r4)
print("size is", r4.size)
print()
```

np_create_ranges.py

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
size is 50

[  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
  95 100]
size is 20

[1.        1.3333333 1.6666666 1.9999999 2.3333332 2.6666665 2.9999998
 3.3333331 3.6666664 3.9999997 4.333333  4.6666663 4.9999996]
size is 13

[1.         1.11111111 1.22222222 1.33333333 1.44444444 1.55555556
 1.66666667 1.77777778 1.88888889 2.        ]
size is 10
```

Working with arrays

- Use normal math operators (+, -, /, and *)
- Use NumPy's builtin functions
- By default, apply to every element
- Can apply to single axis
- Operations on between two arrays applies operator to pairs of element

The array object is smart about applying functions and operators. A function applied to an array is applied to every element of the array. An operator applied to two arrays is applied to corresponding elements of the two arrays.

In-place operators (**+=**, *****=, etc) efficiently modify the array itself, rather than returning a new array.

Example

np_basic_array_ops.py

```
import numpy as np

a = np.array(
    [
        [5, 10, 15],
        [2, 4, 6],
        [3, 6, 9, ],
    ]
)  # create 2D array

b = np.array(
    [
        [10, 85, 92],
        [77, 16, 14],
        [19, 52, 23],
    ]
)  # create another 2D array
print("a")
print(a)
print()
print("b")
print(b)
print()

print("a * 10")
print(a * 10)  # multiply every element by 10 (not in place)
print()

print("a + b")
print(a + b)  # add every element of a to the corresponding element of b
print()

print("b + 3")
print(b + 3)  # add 3 to every element of b
print()

print(f"a.sum(): {a.sum()}")
print(f"a.std(): {a.std()}")
print(f"a.mean(): {a.mean()}")
print(f"a.cumsum(): {a.cumsum()}")
print(f"a.cumprod(): {a.cumprod()}")


def c2f(cel):  # user-defined function
    return (9/5 * cel) + 32

f_temps = c2f(a)  # apply function to elements of a
print("f_temps:\n", f_temps)

print()
a += 1000  # add 1000 to every element of a (in place)
print("a after 'a += 1000'")
print(a)
```

np_basic_array_ops.py

```
a
[[ 5 10 15]
 [ 2  4  6]
 [ 3  6  9]]

b
[[10 85 92]
 [77 16 14]
 [19 52 23]]

a * 10
[[ 50 100 150]
 [ 20  40  60]
 [ 30  60  90]]
```

```
a + b
[[ 15  95 107]
 [ 79  20  20]
 [ 22  58  32]]

b + 3
[[13 88 95]
 [80 19 17]
 [22 55 26]]

a.sum(): 60
a.std(): 3.8297084310253524
a.mean(): 6.666666666666667
a.cumsum(): [ 5 15 30 32 36 42 45 51 60]
a.cumprod(): [      5      50     750    1500    6000   36000  108000  648000 5832000]
f_temps:
 [[41.  50.  59. ]
 [35.6 39.2 42.8]
 [37.4 42.8 48.2]]

a after 'a += 1000'
[[1005 1010 1015]
 [1002 1004 1006]
 [1003 1006 1009]]
```

Shapes

- Number of elements on each axis
- array.shape has shape tuple
- Assign to array.shape to change
- Convert to one dimension
    - array.ravel()
    - array.flatten()
- array.transpose() to flip the shape

Every array has a shape, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the shape attribute of an array. To change the shape of an array, assign to the shape attribute.

The `ravel()` and `flatten()` methods will flatten any array into a single dimension. `ravel()` returns a "view" of the original array, while `flatten()` returns a new array. If you modify the result of ravel(), it will modify the original data.

The `transpose()` method will flip shape (x,y) to shape (y,x). It is equivalent to `array.shape = list(reversed(array.shape))`.

Set one element of the shape tuple to -1 to let numpy calculate the number of items.

Example

np_shapes.py

```
import numpy as np

a1 = np.arange(15)  # create 1D array
print("a1 shape", a1.shape)  # get shape
print()

print(a1)
print()

a1.shape = 3, 5  # reshape to 3x5
print(a1)
print()

a1.shape = 5, 3  # reshape to 5x3
print(a1)
print()

a1.shape = 3, -1  # reshape back to 3x5, let numpy calculate other dimension
print(a1)
print()

print(a1.flatten())  # print array as 1D (always returns a new array())
print()

print(a1.ravel()) # like .flatten(), but makes a *view* if possible (tries not to copy)
print()

print(a1.transpose())  # print transposed array
print("------------------")

a2 = np.arange(40)  # create 1D array
a2.shape = 2, 5, 4  # reshape to 2x5x4

print(a2)
print()
```

np_shapes.py

```
a1 shape (15,)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

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

 [[20 21 22 23]
  [24 25 26 27]
  [28 29 30 31]
  [32 33 34 35]
  [36 37 38 39]]]
```

Selecting data

- Simple indexing similar to lists
  - ARRAY[row, column]
- Slicing
  - start, stop, step
  - start is INclusive, stop is Exclusive

NumPy arrays can be indexed like regular Python lists, but with some convenient extensions. In addition to `ARRAY[row][column]`, NumPy arrays can be indexed with `ARRAY[row-spec,col-spec]`. The row spec can be an integer or a slice.

Slice notation is `start:stop:step` as usual.

For more examples, with visual explanations, see https://solothought.com/tutorial/python-numpy/

Example

np_indexing.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  # sample data

print("a:")
print(a)
print()

print('a[0] =>', a[0])  # first row
print('a[0][0] =>', a[0][0])  # first element of first row
print('a[0,0] =>', a[0,0])  # same, but numpy style
print('a[0,:3] =>', a[0,:3])  # first 3 elements of first row
print()

print('a[:,:2] =>\n', a[:,:2])  # first 2 columns
print()

print('a[:3,0] =>', a[:3,0])  # first column of first 3 rows
print()

print('a[:3, :3] =>\n', a[:3,:3])  # first 3 rows, first 3 columns
print()

print('a[::2] =>\n', a[::2])  # every second row
print()
```

np_indexing.py

```
a:
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

a[0] => [70 31 21 76 19  5 54 66]
a[0][0] => 70
a[0,0] => 70
a[0,:3] => [70 31 21]

a[:,:2] =>
 [[70 31]
 [23 29]
 [11 84]
 [25 13]
 [75 37]]

a[:3,0] => [70 23 11]

a[:3, :3] =>
 [[70 31 21]
 [23 29 71]
 [11 84  7]]

a[::2] =>
 [[70 31 21 76 19  5 54 66]
 [11 84  7 10 31 50 11 98]
 [75 37 11 62 35 76 38  4]]
```

Indexing with Booleans

- Apply relational expression to array
- Result is array of Booleans
- Booleans can be used to index original array

If a relational expression (>, <, >=, ⇐) is applied to an array, the result is a new array containing Booleans reflecting whether the expression was true for each element. That is, for each element of the original array, the resulting array is set to True if the expression is true for that element, and False otherwise.

The resulting Boolean array can then be used as an index, to modify just the elements for which the expression was true.

Example

np_bool_indexing.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  # sample data

print('a =>', a, '\n')

i = a > 50  # create Boolean mask
print('i (a > 50) =>', i, '\n')

print('a[i] =>', a[i], '\n')  # print elements of a that are > 50 using mask

print('a[a > 50] =>', a[a > 50], '\n')  # same, but without creating a separate mask

print('a[i].min(), a[i].max() =>', a[i].min(), a[i].max(), '\n')  # min and max values of
result set with values less than 50

a[i] = 0  # set elements with value > 50 to 0
print('a =>', a, '\n')

print("a[a < 15] += 10")
a[a < 15] += 10  # add 10 to elements < 15
print(a, '\n')
```

np_bool_indexing.py

```
a => [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

i (a > 50) => [[ True False False  True False False  True  True]
 [False False  True False False  True  True  True]
 [False  True False False False False False  True]
 [False False False False False  True False  True]
 [ True False False  True False  True False False]]

a[i] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[a > 50] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[i].min(), a[i].max() => 52 98

a => [[ 0 31 21  0 19  5  0  0]
 [23 29  0 12 27  0  0  0]
 [11  0  7 10 31 50 11  0]
 [25 13 43  1 31  0 41  0]
 [ 0 37 11  0 35  0 38  4]]

a[a < 15] += 10
[[10 31 21 10 19 15 10 10]
 [23 29 10 22 27 10 10 10]
 [21 10 17 20 31 50 21 10]
 [25 23 43 11 31 10 41 10]
 [10 37 21 10 35 10 38 14]]
```

Selecting rows based on conditions

- Index with boolean expressions
- Use **&**, not **and**

To select rows from an array, based on conditions, you can index the array with two or more Boolean expressions.

Since the Boolean expressions return arrays of True/False values, use the **&** bitwise AND operator (or **|** for OR).

Any number of conditions can be applied this way.

```
new_array = old_array[bool_expr1 & bool_expr2 ...]
```

Example

np_select_rows.py

```
import numpy as np

sample_data = np.loadtxt(   # Read some data into 2d array
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

print("first 5 rows of sample_data:")
print(sample_data[:5, :], '\n')

selected = sample_data[  # Index into the existing data
    (sample_data[:, 0] < 10) &  # Combine two Boolean expressions with &
    (sample_data[:, -1] > 35)
]

print("selected")
print(selected)
```

np_select_rows.py

```
first 5 rows of sample_data:
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 [29.  8. 40. 59. 10. 26.]
 [54.  9. 68.  4. 16. 21.]]

selected
[[ 8. 49.  2. 40. 50. 36.]
 [ 4. 49. 39. 50. 23. 39.]
 [ 6.  7. 40. 56. 31. 38.]
 [ 6.  1. 44. 55. 49. 36.]
 [ 5. 22. 45. 49. 10. 37.]]
```

Stacking

- Combining 2 arrays vertically or horizontally
- use `vstack()` or `hstack()`
- Arrays must have compatible shapes

You can combine two or more arrays vertically or horizontally with the `vstack()` or `hstack()` functions. These functions are also handy for adding rows or columns with the results of operations.

Example

np_stacking.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73]]
)  # sample array a

b = np.array(
    [[11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90]]
)  # sample array b

print('a =>\n', a)
print()
print('b =>\n', b)
print()
print('vstack((a,b)) =>\n', np.vstack((a, b)))  # stack arrays vertically (like pancakes)
print()

print('vstack((a,a[0] + a[1])) =>\n', np.vstack((a, a[0] + a[1])))  # add a row with sums
of first two rows
print()

print('hstack((a,b)) =>\n', np.hstack((a, b)))  # stack arrays horizontally (like books
on a shelf)
print()

# add column with product of last two columns
print(
    'np.hstack((a, np.prod(a[:,-2:], axis=1).reshape(2,1))) =>\n',
    np.hstack((a, np.prod(a[:,-2:], axis=1).reshape(2,1)))
)
```

np_stacking.py

```
a =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
 [[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,b)) =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,a[0] + a[1])) =>
 [[ 70  31  21  76  19   5  54  66]
 [ 23  29  71  12  27  74  65  73]
 [ 93  60  92  88  46  79 119 139]]

hstack((a,b)) =>
 [[70 31 21 76 19  5 54 66 11 84  7 10 31 50 11 98]
 [23 29 71 12 27 74 65 73 25 13 43  1 31 52 41 90]]

np.hstack((a, np.prod(a[:,-2:], axis=1).reshape(2,1))) =>
 [[  70   31   21   76   19    5   54   66 3564]
 [  23   29   71   12   27   74   65   73 4745]]
```

ufuncs and builtin operators

- Builtin functions for efficiency
- Map over array
- No **for** loops
- Use **vectorize()** for custom ufuncs

In normal Python, you are used to iterating over arrays, especially nested arrays, with a **for** loop. However, for large amounts of data, this is slow. The reason is that the interpreter must do type-checking and lookups for each item being looped over.

NumPy provides *vectorized* operations which are implemented by *ufuncs* — universal functions. ufuncs are implemented in C and work directly on NumPy arrays. When you use a normal math operator (**+ - * /**, etc) on a NumPy array, it calls the underlying ufunc. For instance, `array1 + array2` calls `np.add(array1, array2)`.

There are over 60 ufuncs built into NumPy. These normally return a NumPy array with the results of the operation. Some have options for putting the output into a different object.

The official docs for ufuncs are here:

https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

You can scroll down to the list of available ufuncs.

Table 20. List of NumPy universal functions (ufunc)

Math operations	
`add(x1, x2, /[, out, where, casting, order, …])`	Add arguments element-wise.
`subtract(x1, x2, /[, out, where, casting, …])`	Subtract arguments, element-wise.
`multiply(x1, x2, /[, out, where, casting, …])`	Multiply arguments element-wise.
`divide(x1, x2, /[, out, where, casting, …])`	Returns a true division of the inputs, element-wise.
`logaddexp(x1, x2, /[, out, where, casting, …])`	Logarithm of the sum of exponentiations of the inputs.
`logaddexp2(x1, x2, /[, out, where, casting, …])`	Logarithm of the sum of exponentiations of the inputs in base-2.
`true_divide(x1, x2, /[, out, where, …])`	Returns a true division of the inputs, element-wise.
`floor_divide(x1, x2, /[, out, where, …])`	Return the largest integer smaller or equal to the division of the inputs.
`negative(x, /[, out, where, casting, order, …])`	Numerical negative, element-wise.
`positive(x, /[, out, where, casting, order, …])`	Numerical positive, element-wise.
`power(x1, x2, /[, out, where, casting, …])`	First array elements raised to powers from second array, element-wise.
`remainder(x1, x2, /[, out, where, casting, …])`	Return element-wise remainder of division.
`mod(x1, x2, /[, out, where, casting, order, …])`	Return element-wise remainder of division.
`fmod(x1, x2, /[, out, where, casting, …])`	Return the element-wise remainder of division.
`divmod(x1, x2[, out1, out2], / [[, out, …])`	Return element-wise quotient and remainder simultaneously.
`absolute(x, /[, out, where, casting, order, …])`	Calculate the absolute value element-wise.
`fabs(x, /[, out, where, casting, order, …])`	Compute the absolute values element-wise.
`rint(x, /[, out, where, casting, order, …])`	Round elements of the array to the nearest integer.
`sign(x, /[, out, where, casting, order, …])`	Returns an element-wise indication of the sign of a number.
`heaviside(x1, x2, /[, out, where, casting, …])`	Compute the Heaviside step function.

`conj(x, /[, out, where, casting, order, …])`	Return the complex conjugate, element-wise.
`conjugate(x, /[, out, where, casting, …])`	Return the complex conjugate, element-wise.
`exp(x, /[, out, where, casting, order, …])`	Calculate the exponential of all elements in the input array.
`exp2(x, /[, out, where, casting, order, …])`	Calculate 2**p for all p in the input array.
`log(x, /[, out, where, casting, order, …])`	Natural logarithm, element-wise.
`log2(x, /[, out, where, casting, order, …])`	Base-2 logarithm of x.
`log10(x, /[, out, where, casting, order, …])`	Return the base 10 logarithm of the input array, element-wise.
`expm1(x, /[, out, where, casting, order, …])`	Calculate exp(x) - 1 for all elements in the array.
`log1p(x, /[, out, where, casting, order, …])`	Return the natural logarithm of one plus the input array, element-wise.
`sqrt(x, /[, out, where, casting, order, …])`	Return the non-negative square-root of an array, element-wise.
`square(x, /[, out, where, casting, order, …])`	Return the element-wise square of the input.
`cbrt(x, /[, out, where, casting, order, …])`	Return the cube-root of an array, element-wise.
`reciprocal(x, /[, out, where, casting, …])`	Return the reciprocal of the argument, element-wise.
`gcd(x1, x2, /[, out, where, casting, order, …])`	Returns the greatest common divisor of |x1| and |x2|
`lcm(x1, x2, /[, out, where, casting, order, …])`	Returns the lowest common multiple of |x1| and |x2|
Trigonometric functions These all use radians when an angle is called for. The ratio of degrees to radians is 180°/**pi**.	
`sin(x, /[, out, where, casting, order, …])`	Trigonometric sine, element-wise.
`cos(x, /[, out, where, casting, order, …])`	Cosine element-wise.
`tan(x, /[, out, where, casting, order, …])`	Compute tangent element-wise.
`arcsin(x, /[, out, where, casting, order, …])`	Inverse sine, element-wise.
`arccos(x, /[, out, where, casting, order, …])`	Trigonometric inverse cosine, element-wise.
`arctan(x, /[, out, where, casting, order, …])`	Trigonometric inverse tangent, element-wise.

`arctan2(x1, x2, /[, out, where, casting, …])`	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
`hypot(x1, x2, /[, out, where, casting, …])`	Given the “legs” of a right triangle, return its hypotenuse.
`sinh(x, /[, out, where, casting, order, …])`	Hyperbolic sine, element-wise.
`cosh(x, /[, out, where, casting, order, …])`	Hyperbolic cosine, element-wise.
`tanh(x, /[, out, where, casting, order, …])`	Compute hyperbolic tangent element-wise.
`arcsinh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic sine element-wise.
`arccosh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic cosine, element-wise.
`arctanh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic tangent element-wise.
`deg2rad(x, /[, out, where, casting, order, …])`	Convert angles from degrees to radians.
`rad2deg(x, /[, out, where, casting, order, …])`	Convert angles from radians to degrees.
Bit-twiddling functions These function all require integer arguments and they manipulate the bit-pattern of those arguments.	
`bitwise_and(x1, x2, /[, out, where, …])`	Compute the bit-wise AND of two arrays element-wise.
`bitwise_or(x1, x2, /[, out, where, casting, …])`	Compute the bit-wise OR of two arrays element-wise.
`bitwise_xor(x1, x2, /[, out, where, …])`	Compute the bit-wise XOR of two arrays element-wise.
`invert(x, /[, out, where, casting, order, …])`	Compute bit-wise inversion, or bit-wise NOT, element-wise.
`left_shift(x1, x2, /[, out, where, casting, …])`	Shift the bits of an integer to the left.
`right_shift(x1, x2, /[, out, where, …])`	Shift the bits of an integer to the right.
Comparison functions[1]	
`greater(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 > x2) element-wise.
`greater_equal(x1, x2, /[, out, where, …])`	Return the truth value of (x1 >= x2) element-wise.
`less(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 < x2) element-wise.
`less_equal(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 =< x2) element-wise.

`not_equal(x1, x2, /[, out, where, casting, …])`	Return (x1 != x2) element-wise.
`equal(x1, x2, /[, out, where, casting, …])`	Return (x1 == x2) element-wise.
`logical_and(x1, x2, /[, out, where, …])`[2]	Compute the truth value of x1 AND x2 element-wise.
`logical_or(x1, x2, /[, out, where, casting, …])`	Compute the truth value of x1 OR x2 element-wise.
`logical_xor(x1, x2, /[, out, where, …])`	Compute the truth value of x1 XOR x2, element-wise.
`logical_not(x, /[, out, where, casting, …])`	Compute the truth value of NOT x element-wise.
`maximum(x1, x2, /[, out, where, casting, …])`	Element-wise maximum of array elements.
`minimum(x1, x2, /[, out, where, casting, …])`	Element-wise minimum of array elements.
`fmax(x1, x2, /[, out, where, casting, …])`	Element-wise maximum of array elements.
`fmin(x1, x2, /[, out, where, casting, …])`	Element-wise minimum of array elements.
Floating functions These all work element-by-element over an array, returning an array output. The description details only a single operation.	
`isfinite(x, /[, out, where, casting, order, …])`	Test element-wise for finiteness (not infinity or not Not a Number).
`isinf(x, /[, out, where, casting, order, …])`	Test element-wise for positive or negative infinity.
`isnan(x, /[, out, where, casting, order, …])`	Test element-wise for NaN and return result as a boolean array.
`isnat(x, /[, out, where, casting, order, …])`	Test element-wise for NaT (not a time) and return result as a boolean array.
`fabs(x, /[, out, where, casting, order, …])`	Compute the absolute values element-wise.
`signbit(x, /[, out, where, casting, order, …])`	Returns element-wise True where signbit is set (less than zero).
`copysign(x1, x2, /[, out, where, casting, …])`	Change the sign of x1 to that of x2, element-wise.
`nextafter(x1, x2, /[, out, where, casting, …])`	Return the next floating-point value after x1 towards x2, element-wise.
`spacing(x, /[, out, where, casting, order, …])`	Return the distance between x and the nearest adjacent number.
`modf(x[, out1, out2], / [[, out, where, …])`	Return the fractional and integral parts of an array, element-wise.
`ldexp(x1, x2, /[, out, where, casting, …])`	Returns x1 * 2**x2, element-wise.

`frexp(x[, out1, out2], / [[, out, where, …])`	Decompose the elements of x into mantissa and twos exponent.
`fmod(x1, x2, /[, out, where, casting, …])`	Return the element-wise remainder of division.
`floor(x, /[, out, where, casting, order, …])`	Return the floor of the input, element-wise.
`ceil(x, /[, out, where, casting, order, …])`	Return the ceiling of the input, element-wise.
`trunc(x, /[, out, where, casting, order, …])`	Return the truncated value of the input, element-wise.

[1]Warning — do not use the Python keywords **and** and **or** to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators **&** and **|** instead.

[2]Warning — bit-wise operators **&** and **|** are the proper way to perform element-by-element comparisons. Be sure you understand the operator precedence: `(a > 2) & (a < 5)` instead of `a > 2 & a < 5`.

Vectorizing functions

- Many functions "just work"
- `np.vectorize()` allows user-defined function to be broadcast.

ufuncs will automatically be broadcast across any array to which they are applied. For user-defined functions that don’t correctly broadcast, NumPy provides the **vectorize()** function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

np_vectorize.py

```
import time
import numpy as np

sample_data = np.loadtxt(   # Create some sample data
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

def set_default(value, limit, default): # Define function with more than one parameter
    if value > limit:
        value = default

    return value

MAX_VALUE = 50      # Define max value
DEFAULT_VALUE = -1  # Define default value

print("Version 1: looping over arrays")
start = time.perf_counter() # Get the current time as Unix timestamp (large float)
try:
    version1_array = np.zeros(sample_data.shape, dtype=int)  # Create array to hold
results
    for i, row in enumerate(sample_data):  # Iterate over rows and columns of input array
        for j, column in enumerate(row):
            version1_array[i, j] = set_default(sample_data[i, j], MAX_VALUE,
DEFAULT_VALUE)   # Call function and put result in new array
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()  # Get current time
    elapsed = end - start  # Get elapsed number of seconds and print them out
    print(version1_array)
    print(f"took {elapsed:.5f} seconds")
finally:
    print()


print("Version 2: broadcast without vectorize()")
start = time.perf_counter()
try:
    print("Without sp.vectorize:")
    version2_array = set_default(sample_data, MAX_VALUE, DEFAULT_VALUE) # Pass array to
function; it fails because it has more than one parameter
```

```
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()
    elapsed = end - start
    print(version2_array)
    print(f"took {elapsed:.5f} seconds")
finally:
    print()

print("Version 3: broadcast with vectorize()")
set_default_vect = np.vectorize(set_default)  # Convert function to vectorized version --
creates function that takes one parameter and has the other two "embedded" in it

start = time.perf_counter()
try:
    print("With sp.vectorize:")
    version3_array = set_default_vect(sample_data, MAX_VALUE, DEFAULT_VALUE) # Call
vectorized version with same parameters
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()
    elapsed = end - start
    print(version3_array)
    print(f"took {elapsed:.5f} seconds")
finally:
    print()
```

np_vectorize.py

```
Version 1: looping over arrays
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00362 seconds

Version 2: broadcast without vectorize()
Without sp.vectorize:
Function failed: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()

Version 3: broadcast with vectorize()
With sp.vectorize:
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00102 seconds
```

Getting help

- Several help functions
  - `numpy.info()`
  - `numpy.lookfor()`
  - `numpy.source()`

NumPy has several functions for getting help. The first is `numpy.info()`, which provides a brief explanation of a function, class, module, or other object as well as some code examples.

If you’re not sure what function you need, you can try `numpy.lookfor()`, which does a keyword search through the NumPy documentation.

These functions are convenient when using **iPython** or **Jupyter**.

Example

np_info.py

```
import numpy as np
import scipy.fftpack as ff


def main():
    np.info(ff.fft)  # Get help on the fft() function

    print('-' * 60)

    np.source(ff.fft)  # View the source of the fft() function

    print('-' * 60)

    np.lookfor('convolve') # search np docs


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

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use array.flat to iterate through all elements
- Don’t do it unless you have to

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use `array.flat`.

However, iterating over a NumPy array is generally much less efficient than using a *vectorized* approach — calling a *ufunc* or directly applying a math operator. Some tasks may require it, but you should avoid it if possible.

Example

np_iterating.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76],
     [23, 29, 71, 12]]
)  # sample array

print('a =>\n', a)
print()

print("for row in a: =>")
for row in a:  # iterate over rows
    print("row:", row)
print()

print("for column in a.T:")
for column in a.T:  # iterate over columns by transposing the array
    print("column:", column)
print()

print("for elem in a.flat: =>")
for elem in a.flat:  # iterate over all elements (row-major)
    print("element:", elem)
```

np_iterating.py

```
a =>
 [[70 31 21 76]
 [23 29 71 12]]

for row in a: =>
row: [70 31 21 76]
row: [23 29 71 12]

for column in a.T:
column: [70 23]
column: [31 29]
column: [21 71]
column: [76 12]

for elem in a.flat: =>
element: 70
element: 31
element: 21
element: 76
element: 23
element: 29
element: 71
element: 12
```

Matrix Multiplication

- Use normal ndarrays
- Most operations same as ndarray
- Use @ for multiplication

For traditional matrix operations, use a normal ndarray. Most operations are the same as for ndarrays. For matrix (diagonal) multiplication, use the @ (matrix multiplication) operator.

For transposing, use *array*.transpose(), or just *array*.T.

There was formerly a Matrix type in NumPy, but it is deprecated since the addition of the @ operator in Python 3.5

Example

np_matrices.py

```
import numpy as np

m1 = np.array(
    [[2, 4, 6],
     [10, 20, 30]]
)  # sample 2x3 array

m2 = np.array([[1, 15],
                [3, 25],
                [5, 35]])  # sample 3x2 array

print('m1 =>\n', m1)
print()

print('m2 =>\n', m2)
print()

print('m1 * 10 =>\n', m1 * 10)  # multiply every element of m1 times 10
print()

print('m1 @ m2 =>\n', m1 @ m2)  # matrix multiply m1 times m2 -- diagonal product
print()
```

np_matrices.py

```
m1 =>
 [[ 2  4  6]
 [10 20 30]]

m2 =>
 [[ 1 15]
 [ 3 25]
 [ 5 35]]

m1 * 10 =>
 [[ 20  40  60]
 [100 200 300]]

m1 @ m2 =>
 [[  44  340]
 [ 220 1700]]
```

Data Types

- Default is **float**
- Data type is inferred from initialization data
- Can be specified with `arange()`, `ones()`, `zeros()`, etc.

Numpy defines around 30 numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using `arange()`, `ones()`, etc., to create arrays, the **dtype** parameter can be used to specify the data type.

The default data type is **np.float_**, which maps to the Python builtin type **float**.

The data type cannot be changed after an array is created.

See https://numpy.org/devdocs/user/basics.types.html for more details.

Example

np_data_types.py

```
import numpy as np

r1 = np.arange(45)  # create array -- arange() defaults to int
r1.shape = (3, 3, 5)  # create array -- passing float makes all elements float
print('r1 datatype:', r1.dtype)
print('r1 =>\n', r1, '\n')

r2 = np.arange(45.)  # create array -- set datatype to short int
r2.shape = (3, 3, 5)
print('r2 datatype:', r2.dtype)
print('r2 =>\n', r2, '\n')

r3 = np.arange(45, dtype=np.int16)  # create array -- set datatype to short int
r3.shape = (3, 3, 5)
print('r3 datatype:', r3.dtype)
print('r3 =>\n', r3, '\n')
```

np_data_types.py

```
r1 datatype: int64
r1 =>
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]

r2 datatype: float64
r2 =>
 [[[ 0.  1.  2.  3.  4.]
  [ 5.  6.  7.  8.  9.]
  [10. 11. 12. 13. 14.]]

 [[15. 16. 17. 18. 19.]
  [20. 21. 22. 23. 24.]
  [25. 26. 27. 28. 29.]]

 [[30. 31. 32. 33. 34.]
  [35. 36. 37. 38. 39.]
  [40. 41. 42. 43. 44.]]]

r3 datatype: int16
r3 =>
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]
```

Reading and writing Data

- Read data from files into **ndarray**
- Text files
  - `loadtxt()`
  - `savetxt()`
  - `genfromtxt()`
- Binary (or text) files
  - `fromfile()`
  - `tofile()`

NumPy has several functions for reading data into an array.

`numpy.loadtxt()` reads a delimited text file. There are many options for fine-tuning the import.

`numpy.genfromtxt()` is similar to `numpy.loadtxt()`, but also adds support for handling missing data

Both functions allow skipping rows, user-defined per-column converters, setting the data type, and many others.

To save an array as a text file, use the `numpy.savetxt()` function. You can specify delimiters, header, footer, and formatting.

To read binary data, use `numpy.fromfile()`. It expects a file to contain all the same data type, i.e., ints or floats of a specified type. It will default to floats. `fromfile()` can also be used to read text files.

To save as binary data, you can use `numpy.tofile()`, but **tofile()** and **fromfile()** are not platform-independent. See the next section on **save()** and **load()** for platform-independent I/O.

Example

np_savetxt_loadtxt.py

```
import numpy as np

sample_data = np.loadtxt(   # Load data from space-delimited file
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

print(sample_data)
print('-' * 60)

sample_data  /= 10  # Modify sample data

float_file_name = 'save_data_float.txt'

np.savetxt(float_file_name, sample_data, delimiter=",", fmt="%5.2f")  # Write data to
text file as floats, rounded to two decimal places, using commas as delimiter

int_file_name = 'save_data_int.txt'

np.savetxt(int_file_name, sample_data, delimiter=",", fmt="%d")  # Write data to text
file as ints, using commas as delimiter

data = np.loadtxt(float_file_name, delimiter=",")  # Read data back into ndarray
print(data)
```

np_savetxt_loadtxt.py

```
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 ...
 [26. 20. 54. 46. 38. 23.]
 [ 9.  5. 59. 23.  2. 26.]
 [46. 34. 25.  8. 39. 34.]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Example

np_tofile_fromfile.py

```
import numpy as np

sample_data = np.loadtxt(   # Read in sample data
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

sample_data  /= 10  # Modify sample data

print(sample_data)
print("-" * 60)

file_name = 'sample.dat'

sample_data.tofile(file_name)  # Write data to file (binary, but not portable)

data = np.fromfile(file_name)  # Read binary data from file as one-dimensional array
data.shape = sample_data.shape  # Set shape to shape of original array

print(data)
```

np_tofile_fromfile.py

```
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Saving and retrieving arrays

- Efficient binary format
- Save as NumPy data
  - Use `numpy.save()`
- Read into ndarray
  - Use `numpy.load()`

To save an array as a NumPy data file, use `numpy.save()`. This will write the data out to a specified file name, adding the extension '.npy'.

To read the data back into a NumPy ndarray, use `numpy.load()`. Data are read and written in a way that preserves precision and endianness.

This the most efficient way to store numeric data for later retrieval, compared to **savetext()** and **loadtext()** or **tofile()** and **fromfile()**. Files written with `numpy.save()` are not human-readable.

Example

np_save_load.py

```
import numpy as np

sample_data = np.loadtxt(   # Read some sample data into an ndarray
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=int
)

sample_data  *= 100  # Modify the sample data (multiply every element by 100)

print(sample_data)

file_name = 'sampledata'

np.save(file_name, sample_data)  # Write entire array out to NumPy-format data file (adds
.npy extension)

retrieved_data = np.load(file_name + '.npy')  # Retrieve data from saved file

print('-' * 60)
print(retrieved_data)
```

np_save_load.py

```
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
------------------------------------------------------------
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
```

Chapter 14 Exercises

Exercise 14-1 (big_arrays.py)

Starting with the file big_arrays.py, convert the Python list values into a NumPy array.

Make a copy of the array named values_x_3 with all values multiplied by 3.

Print out values_x_3

Exercise 14-2 (create_range.py)

Using arange(), create an array of 35 elements.

Reshape the arrray to be 5 x 7 and print it out.

Reshape the array to be 7 x 5 and print it out.

Exercise 14-3 (create_linear_space.py)

Using linspace(), create an array of 500 elements evenly spaced between 100 and 200.

Reshape the array into 5 x 10 x 10.

Multiply every element by .5

Print the result.

Chapter 15: 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    1.509755
b    0.763558
c    2.105220
d    0.280344
e   -0.419533
f   -1.353892
g    1.571131
h   -0.058185
i   -1.072925
j   -1.126756
dtype: float64

s2: 0    1.509755
1    0.763558
2    2.105220
3    0.280344
4   -0.419533
5   -1.353892
6    1.571131
7   -0.058185
8   -1.072925
9   -1.126756
dtype: float64

selecting elements
h   -0.058185
b    0.763558
dtype: float64

a    1.509755
b    0.763558
c    2.105220
dtype: float64

slice of elements
b    0.763558
c    2.105220
d    0.280344
dtype: float64

sum(), mean(), min(), max():
2.198718296917599 0.2198718296917599 -1.353891649975827 2.1052203535725016

cumsum(), cumprod():
a    1.509755
```

```
b    2.273313
c    4.378534
d    4.658878
e    4.239345
f    2.885453
g    4.456584
h    4.398399
i    3.325474
j    2.198718
dtype: float64 a    1.509755
b    1.152786
c    2.426868
d    0.680358
e   -0.285432
f    0.386444
g    0.607155
h   -0.035327
i    0.037904
j   -0.042708
dtype: float64

True
False

s3 (which is s1 * 10)
a    15.097552
b     7.635581
c    21.052204
d     2.803439
e    -4.195327
f   -13.538916
g    15.711312
h    -0.581852
i   -10.729249
j   -11.267559
dtype: float64

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

```
dtype: bool

assign -1 where mask is true
a    15.097552
b     7.635581
c    21.052204
d    -1.000000
e    -1.000000
f    -1.000000
g    15.711312
h    -1.000000
i    -1.000000
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 21. 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
```

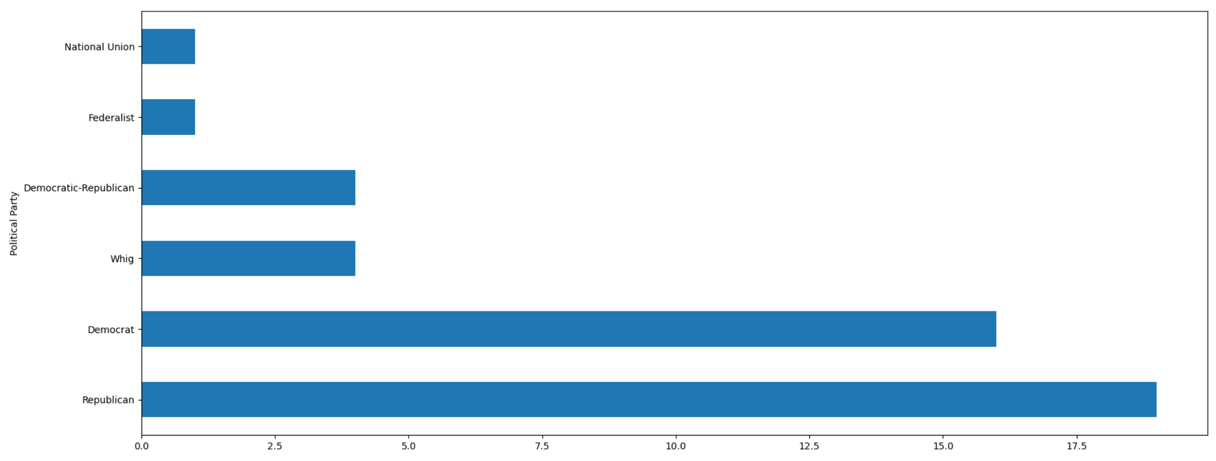


Figure 1. Bar graph of value counts

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 22. 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 23. 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 15 Exercises

Exercise 15-1 (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 15-2 (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 16: 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 16 Exercises

Exercise 16-1 (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.

Appendix A: Field Guide to Python Expressions

Table 24. Python Expressions

Expression	Meaning

`a, b, c`
`(a, b, c)`
	tuple

`a,`
`(a,)`
	tuple with one element

`()`	empty tuple
`[a, b, c]`	list
`[]`	empty list
`{a, b, c}`	set
`{a:r, b:s, c:t}`	dictionary
`{}`	empty dictionary
`[expr for var in iterable if condition]`	list comprehension
`(expr for var in iterable if condition)`	generator expression
`{expr for var in iterable if condition}`	set comprehension
`{key:value for var in iterable if condition}`	dictionary comprehension
`a, b, c = iterable`	iterable unpacking
`a, *b, c = iterable`	extended iterable unpacking
`a if b else c`	conditional expression
`lambda VAR …: VALUE`	lambda function

Appendix B: Python Bibliography

Data Science

- ***Building machine learning systems with Python***. William Richert, Luis Pedro Coelho. Packt Publishing
- ***High Performance Python***. Mischa Gorlelick and Ian Ozsvald. O’Reilly Media
- ***Introduction to Machine Learning with Python***. Sarah Guido. O’Reilly & Assoc.
- ***iPython Interactive Computing and Visualization Cookbook***. Cyril Rossant. Packt Publishing
- ***Learning iPython for Interactive Computing and Visualization***. Cyril Rossant. Packt Publishing
- ***Learning Pandas***. Michael Heydt. Packt Publishing
- ***Learning scikit-learn: Machine Learning in Python***. Raúl Garreta, Guillermo Moncecchi. Packt Publishing
- ***Mastering Machine Learning with Scikit-learn***. Gavin Hackeling. Packt Publishing
- ***Matplotlib for Python Developers***.Sandro Tosi.Packt Publishing
- ***Numpy Beginner’s Guide.Ivan Idris***.Packt Publishing
- ***Numpy Cookbook.Ivan Idris***.Packt Publishing
- ***Practical Data Science Cookbook.Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta***.Packt Publishing
- ***Python Text Processing with NLTK 2.0 Cookbook.Jacob Perkins***.Packt Publishing
- ***Scikit-learn cookbook.Trent Hauck***.Packt Publishing
- ***Python Data Visualization Cookbook.Igor Milovanovic***.Packt Publishing
- ***Python for Data Analysis.Wes McKinney.***. O’Reilly & Assoc

Design Patterns

- ***Design Patterns: Elements of Reusable Object-Oriented Software.Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides***.Addison-Wesley Professional
- ***Head First Design Patterns.Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra***.O’Reilly Media
- ***Learning Python Design Patterns.Gennadiy Zlobin***.Packt Publishing
- ***Mastering Python Design Patterns.Sakis Kasampalis***.Packt Publishing

General Python development

- ***Expert Python Programming.Tarek Ziadé***.Packt Publishing
- ***Fluent Python.Luciano Ramalho***. O’Reilly & Assoc.

- ***Learning Python, 2nd Ed..Mark Lutz, David Asher***. O'Reilly & Assoc.
- ***Mastering Object-oriented Python.Stephen F. Lott***.Packt Publishing
- ***Programming Python, 2nd Ed. .Mark Lutz***. O'Reilly & Assoc.
- ***Python 3 Object Oriented Programming.Dusty Phillips***.Packt Publishing
- ***Python Cookbook, 3rd. Ed.. David Beazley, Brian K. Jones***. O'Reilly & Assoc.
- ***Python Essential Reference, 4th. Ed..David M. Beazley***.Addison-Wesley Professional
- ***Python in a Nutshell.Alex Martelli***. O'Reilly & Assoc.
- ***Python Programming on Win32.Mark Hammond, Andy Robinson***. O'Reilly & Assoc.
- ***The Python Standard Library By Example.Doug Hellmann***.Addison-Wesley Professional

Misc

- ***Python Geospatial Development.Erik Westra***.Packt Publishing
- ***Python High Performance Programming.Gabriele Lanaro***.Packt Publishing

Networking

- ***Python Network Programming Cookbook.Dr. M. O. Faruque Sarker***.Packt Publishing
- ***Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers.T J O'Connor***.Syngress
- ***Web Scraping with Python.Ryan Mitchell***.O'Reilly & Assoc.

Testing

- ***Python Testing Cookbook.Greg L. Turnquist***.Packt Publishing
- ***Learning Python Testing.Daniel Arbuckle***.Packt Publishing
- ***Learning Selenium Testing Tools, 3rd Ed. .Raghavendra Prasad MG***.Packt Publishing

Web Development

- ***Building Web Applications with Flask.Italo Maia***.Packt Publishing
- ***Django 1.0 Website Development.Ayman Hourieh***.Packt Publishing
- ***Django 1.1 Testing and Development.Karen M. Tracey***.Packt Publishing
- ***Django By Example.Antonio Melé***.Packt Publishing
- ***Django Design Patterns and Best Practices.Arun Ravindran***.Packt Publishing
- ***Django Essentials.Samuel Dauzon***.Packt Publishing

- ***Django Project Blueprints.Asad Jibran Ahmed***.Packt Publishing
- ***Flask Blueprints.Joel Perras***.Packt Publishing
- ***Flask by Example.Gareth Dwyer***.Packt Publishing
- ***Flask Framework Cookbook.Shalabh Aggarwal***.Packt Publishing
- ***Flask Web Development.Miguel Grinberg***. O'Reilly & Assoc.
- ***Full Stack Python (e-book only).Matt Makai***.Gumroad (or free download)
- ***Full Stack Python Guide to Deployments (e-book only).Matt Makai***.Gumroad (or free download)
- ***High Performance Django.Peter Baumgartner, Yann Malet***.Lincoln Loop
- ***Instant Flask Web Development.Ron DuPlain***.Packt Publishing
- ***Learning Flask Framework.Matt Copperwaite, Charles O Leifer***.Packt Publishing
- ***Mastering Flask.Jack Stouffer***.Packt Publishing
- ***Two Scoops of Django 3.X: Best Practices for the Django Web Framework. Daniel Roy Greenfeld, Audrey Roy Greenfeld***.Two Scoops Press
- ***Web Development with Django Cookbook.Aidas Bendoraitis***.Packt Publishing

Index