Supplemental Intro to Python Material

John Strickler

Version 1.1, February 2024

Table of Contents

Chapter 1: Unit Testing with pytest . . . 1
- What is a unit test? . . . 2
- The pytest module . . . 3
- Creating tests . . . 4
- Running tests (basics) . . . 5
- Special assertions . . . 7
- Fixtures . . . 10
- User-defined fixtures . . . 11
- Builtin fixtures . . . 14
- Configuring fixtures . . . 18
- Parametrizing tests . . . 21
- Marking tests . . . 24
- Running tests (advanced) . . . 26
- Skipping and failing . . . 28
- Mocking data . . . 31
- Mocking in pytest . . . 32
- Pytest plugins . . . 37

Chapter 2: IPython and JupyterLab . . . 39
- About IPython . . . 40
- Starting IPython . . . 41
- Getting Help . . . 42
- IPython features . . . 43
- Tab Completion . . . 44
- Magic Commands . . . 45
- Loading and running Python scripts . . . 46
- External commands . . . 48
- Using history . . . 49
- Saving sessions . . . 51
- Using Pastebin . . . 52
- Benchmarking . . . 53
- Profiles . . . 54
- Jupyter notebooks . . . 58
- JupyterLab Demo . . . 58
- For more information . . . 59

Chapter 3: Introduction to NumPy . . . 60
- Python’s scientific stack . . . 61

NumPy overview . . . 62
Creating Arrays . . . 63
Creating ranges . . . 67
Working with arrays . . . 69
Shapes . . . 72
Selecting data . . . 75
Indexing with Booleans . . . 78
Selecting rows based on conditions . . . 81
Stacking . . . 83
ufuncs and builtin operators . . . 86
Vectorizing functions . . . 92
Getting help . . . 96
Iterating . . . 97
Matrix Multiplication . . . 100
Data Types . . . 103
Reading and writing Data . . . 106
Saving and retrieving arrays . . . 110
Chapter 4: Introduction to Pandas . . . 113
About pandas . . . 114
Tidy data . . . 115
pandas architecture . . . 116
Series . . . 117
DataFrames . . . 123
Reading Data . . . 127
Data summaries . . . 132
Basic Indexing . . . 136
Saner indexing with .loc, .iloc, and .at . . . 140
Broadcasting . . . 149
Counting unique occurrences . . . 152
Creating new columns . . . 153
Removing entries . . . 155
Useful pandas methods . . . 159
Even more pandas … . . . 161
Chapter 5: Introduction to Matplotlib . . . 163
About matplotlib . . . 164
matplotlib architecture . . . 165
Matplotlib Terminology . . . 166
Matplotlib Keeps State . . . 167

What Else Can You Do? 168
Matplotlib Demo 169
Index 171

Chapter 1: 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.

Each test supplies one or more **assertions**. An assertion confirms that some condition is true.

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 *or* a module than contains unit tests (tests can be grouped with *markers*).

fixture
: A special parameter of a unit test 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.

[1] The builtin unit testing module, **unittest**, *is* based on **xUnit** patterns, as implemented in Java and other languages. The `pytest`` builtin test runner will detect Unittest-based 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 should use the builtin **assert** statement one or more times to confirm that the test passes. If the assertion fails, the test fails.

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

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).

```
assert result == 'spam'
assert 2 == 3, "Two is not equal to three!"
```

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

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.

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 …`

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**.

Making test files executable

While you should normally use the test runner, `pytest`, 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.

Example

tests/test_simple.py

```python
import pytest

def test_two_plus_two_equals_four():  # tests should begin with "test" (or will not be found automatically)
    assert 2 + 2 == 4   # if assert statement succeeds, the test passes

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

pytest -v tests/test_simple.py OR python tests/test_simple.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 1 item

tests/test_simple.py::test_two_plus_two_equals_four PASSED

============================== 1 passed in 0.01s ===============================
```

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

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


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

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

tests/test_special_assertions.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 3 items

tests/test_special_assertions.py::test_missing_filename PASSED
tests/test_special_assertions.py::test_list PASSED
tests/test_special_assertions.py::test_approximate_pi PASSED

============================== 3 passed in 0.02s ===============================
```

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

```python
from collections import namedtuple
import pytest
import sqlite3
import os

Person = namedtuple('Person', 'first_name last_name')  # create object to test

FIRST_NAME = "Guido"
LAST_NAME = "Von Rossum"

THIS_DIR = os.path.dirname(os.path.abspath(__file__))
president_db_path = os.path.join(THIS_DIR, 'presidents.db')

db_conn = sqlite3.connect(president_db_path)  # open relative to EXAMPLES
db_cursor = db_conn.cursor()
db_cursor.row_factory = sqlite3.Row  # set the row factory to be a Row object

@pytest.fixture

def presidents():
    db_cursor.execute('select * from presidents')
    return db_cursor.fetchall()

@pytest.fixture  # mark person as a fixture
def person():
    """
    Return a 'Person' named tuple with fields 'first_name' and 'last_name'
    """
    return Person(FIRST_NAME, LAST_NAME)  # return value of fixture

def test_first_name(person):  # pass fixture as test parameter
    assert person.first_name == FIRST_NAME

def test_last_name(person):  # pass fixture as test parameter
    assert person.last_name == LAST_NAME

def test_john_tyler_is_from_virginia(presidents):
    assert presidents[9]['birthstate'] == 'Virginia'  # John Tyler is 10th president

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

tests/test_simple_fixture.py

```
============================ test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 3 items

tests/test_simple_fixture.py::test_first_name PASSED
tests/test_simple_fixture.py::test_last_name PASSED
tests/test_simple_fixture.py::test_john_tyler_is_from_virginia PASSED

============================== 3 passed in 0.02s ===============================
```

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("Counter 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("Counter 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
    print("STDOUT:", out)

def bhello():
    print(b"Hello, binary pytesting world\n")

def test_capsysbinary(capsysbinary):
    bhello()  # Call function that writes binary text to STDOUT
    out, err = capsysbinary.readouterr()  # Get captured output
    print("BINARY STDOUT:", out)

def test_temp_dir1(tmpdir):
    print("TEMP DIR:", str(tmpdir))  # tmpdir fixture provides unique temporary folder
name

def test_temp_dir2(tmpdir):
    print("TEMP DIR:", str(tmpdir))

def test_temp_dir3(tmpdir):
    print("TEMP DIR:", str(tmpdir))

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

tests/test_builtin_fixtures.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 6 items

tests/test_builtin_fixtures.py::test_cache Counter before: 30
Counter after: 31
PASSED
tests/test_builtin_fixtures.py::test_capsys STDOUT: Hello, pytesting world

PASSED
tests/test_builtin_fixtures.py::test_capsysbinary BINARY STDOUT: b"b'Hello, binary
pytesting world\\n'\n"
PASSED
tests/test_builtin_fixtures.py::test_temp_dir1 TEMP DIR:
/private/var/folders/p7/_ryqngjd3jn_ppndvnhdzqch0000gn/T/pytest-of-jstrick/pytest-
0/test_temp_dir10
PASSED
tests/test_builtin_fixtures.py::test_temp_dir2 TEMP DIR:
/private/var/folders/p7/_ryqngjd3jn_ppndvnhdzqch0000gn/T/pytest-of-jstrick/pytest-
0/test_temp_dir20
PASSED
tests/test_builtin_fixtures.py::test_temp_dir3 TEMP DIR:
/private/var/folders/p7/_ryqngjd3jn_ppndvnhdzqch0000gn/T/pytest-of-jstrick/pytest-
0/test_temp_dir30
PASSED

============================= 6 passed in 0.03s ==============================
```


Table 1. 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

@fixture
def common_fixture():  # user-defined fixture
    return ['alpha', 'beta', 'gamma']


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

Example

tests/test_config.py

```
import pytest


def test_stdout():  # unit test that writes to STDOUT
    print("WHOOPEE", end=" ")
    assert 1


def test_two(common_fixture):   # unit test that uses fixture from conftest.py
    assert "alpha" in common_fixture
    assert "beta" in common_fixture
    assert "gamma" in common_fixture


if __name__ == '__main__':
    pytest.main([__file__, "-s"])   # run tests (without stdout/stderr capture) when this
script is run
```

tests/test_config.py

```
============================ test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collected 2 items

tests/test_config.py WHOOPEE ..

============================== 2 passed in 0.01s ===============================
```


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


def triple(x):  # Function to test
    return x * 3

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

@pytest.mark.parametrize("input,result", test_data)  # Parametrize the test with the test data; the first argument is a string defining parameters to the test and mapping them to the test data
def test_triple(input, result):  # The test expects two parameters (which come from each element of test data)
    print("input {} result {}:".format(input, result))  # The test expects two parameters (which come from each element of test data)
    assert triple(input) == result   # Test the function with the parameters


if __name__ == "__main__":
    pytest.main([__file__, '-s'])
```


tests/test_parametrization.py

```
============================ test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collected 3 items

tests/test_parametrization.py input 5 result 15:
.input a result aaa:
.input [True] result [True, True, True]:
.

============================= 3 passed in 0.01s ===============================
```

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.alpha  # Mark with label alpha
def test_two():
    assert 1

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

if __name__ == '__main__':
    pytest.main([__file__, '-m alpha'])  # Only tests marked with alpha will run
(equivalent to 'pytest -m alpha' on command line)
```

tests/test_mark.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collected 3 items / 1 deselected / 2 selected

tests/test_mark.py ..                                                   [100%]

====================== 2 passed, 1 deselected in 0.02s =======================
```

Running tests (advanced)

- Run all tests
- Run by
  - function
  - class
  - module
  - name match
  - group

pytest provides many ways to select which tests to run.

Running all tests

To run all tests in the current and any descendent directories, use

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 module, 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
```


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'*

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 testing 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


if __name__ == '__main__':
    pytest.main([__file__, '-vs'])
```

tests/test_skip.py

```
============================ test session starts =============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 5 items

tests/test_skip.py::test_one PASSED
tests/test_skip.py::test_two SKIPPED (can not currently test)
tests/test_skip.py::test_three SKIPPED (only implemented on Windows)
tests/test_skip.py::test_four XPASS
tests/test_skip.py::test_five XFAIL

============== 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.05s ==============
```


Mocking data

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

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

The solution is to use a **mock** object, which pretends 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.

A *stub* is an object that returns minimal information, and is also useful in testing. However, a mock object is more elaborate, with record/playback capability, assertions, and other features.

Mocking in pytest

- Use pytest-mock plugin
    - Can also use unittest.mock.Mock
- Emulate resources

pytest can use **unittest.mock**, from the standard library, or the **pytest-mock** plugin, which provides a wrapper around unittest.mock

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

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

The second is to monkey-patch a library, 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.

Installing the modules to test

Before running `test_mock.py` and `test_mock_pymock.py`, `spamlib` and `hamlib` must be installed; otherwise tests won't be able to import them. You can install them in editable mode:

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_value * 10)
    s = Spam(ham_value)  # Create instance of Spam, which calls ham()
    assert s.value == ham_result
    assert spamlib.spam.ham.calledoncewith(ham_value)


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

tests/test_mock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 1 item

tests/test_mock.py::test_spam_calls_ham PASSED

============================== 1 passed in 0.01s ===============================
```

Example

tests/test_mock_pymock.py

```python
import pytest  # Needed for test runner
from spamlib import spam

SEARCH_TERM = 'bug'
SEARCH_STRING = '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(SEARCH_TERM, SEARCH_STRING)  # 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(SEARCH_TERM, SEARCH_STRING)


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

tests/test_mock_pymock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... collected 1 item

tests/test_mock_pymock.py::test_spam_search_calls_re_search PASSED

============================== 1 passed in 0.02s ===============================
```

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'

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

tests/test_mock_play.py

```
mock calls: [call.spam('a', 'b'), call.ham('wombat'), call.eggs(1, 2, 3)]
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/common/examples/tests
configfile: pytest.ini
plugins: anyio-4.0.0, mock-3.12.0, django-4.5.2
collecting ... mock calls: [call.spam('a', 'b'), call.ham('wombat'), call.eggs(1, 2, 3)]
collected 1 item

tests/test_mock_play.py::test_m1_returns_correct_list PASSED

============================== 1 passed in 0.01s ===============================
```


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 1 Exercises

Exercise 1-1 (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 2: IPython and JupyterLab

Objectives

- Learn the basics of IPython
- Apply magics
- List and replay commands
- Run external commands
- Create profiles
- Use Jupyter notebooks in JupyterLab

About IPython

- Enhanced python interpreter
- Great for "playing around" with Python
- Saves running entire script
- Not intended for application development
- Embedded in Jupyter notebooks

IPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It is very handy for quickly trying out Python features or for casual data analysis.

Command line interface

When started from a command line, starts a read-execute-print loop (REPL), also known as an interactive interpreter.

Ipython uses different colors for variables, functions, strings, comments, and so forth.

Jupyter notebook

IPython also provides a *kernel* for embedding IPython in Jupyter notebooks. A Jupyter notebook is a web-based interface consisting of *cells*, which contain code or documentation. Jupyter notebooks can run code from many different languages in addition to Python.

JupyterLab

This is the web-based interface to manage multiple Jupyter notebooks, as well as images, terminal prompts, and other resources.

Starting IPython

- Type `ipython` at the command line
- Huge number of options

To get started with IPython

- Type `ipython` at the command line

OR

- Double-click the IPython icon from Windows explorer.

IPython works like the normal interactive Python interpreter, but with many more features.

There is a huge number of options. To see them all, invoke IPython with the --help-all option:

```
ipython --help-all
```

Use the `--colors=NoColor` option to turn off syntax highlighting and other colorized features.

Getting Help

- ? basic help
- %quickref quick reference
- help standard Python help
- thing? help on thing

IPython provides help in several ways.

Typing `?` at the prompt will display an introduction to IPython and a feature overview.

For a quick reference card, type `%quickref`.

To start Python's normal help system, type `help`.

For help on any Python object, type `object?` or `?object`. This is similar to saying `help("object")` in the default interpreter, but is "smarter".

> For more help, add a second question mark. This does not work for all objects, however, and sometimes it displays the source code of the module containing the object definition.

IPython features

- Name completion (variables, modules, methods, folders, files, etc.)
- Enhanced help system
- Autoindent
- Syntax highlighting
- 'Magic' commands for controlling IPython itself
- Easy access to shell commands
- Dynamic introspection (dir() on steroids)
- Search namespaces with wildcards
- Commands are numbered (and persistent) for recall
- Aliasing system for interpreter commands
- Simplified (and lightweight) persistence
- Session logging (can be saved as scripts)
- Detailed tracebacks when errors occur
- Session restoring (playback log to specific state)
- Flexible configuration system
- Easy access to Python debugger
- Simple profiling
- Interactive parallel computing (if supported by hardware)
- Background execution in separate thread
- Auto-parentheses ('sin 3' becomes 'sin(3)'
- Auto-quoting (',foo a b' becomes 'foo("a","b")'

Tab Completion

- Press `Tab` to complete
  - keywords
  - variables
  - modules
  - methods and attributes
  - parameters to functions
  - file and directory names

Pressing `Tab` will invoke **tab completion**, AKA **autocomplete**. If there is only one possible completion, it will be expanded. If there is more than one completion that will match, IPython will display a list of possible completions.

Autocomplete works on keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Magic Commands

- Start with `%` (line magic) or `%%` (cell magic)
- Simplify common tasks
- Use `%lsmagic` to list all magic commands

One of the enhancements in IPython is the set of "magic" commands. These are meta-commands (macros) that help you manipulate the IPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, `%history` will list previous commands.

Type `lsmagic` for a list of all magics

If the magic command is not the same as a name in your Python code, you can leave off the leading `%` or `%%`.

Loading and running Python scripts

- Run script in current session
- `%run` runs script
- `%load` loads script source code into IPython

IPython provides two magics to run scripts — one to run directly, and one to run indirectly. Both will run the script in the context of the current IPython session.

Running scripts directly

The `%run` magic just takes a script name, and runs it. This method does not allow IPython magics to be executed as part of a script.

```
In [1]: %run ../EXAMPLES/my_vars.py
```

```
In [2]:  user_name
Out[2]: 'Susan'
```

```
In [3]: snake
Out[3]: 'Eastern Racer'
```


Running scripts indirectly

The `%load` magic takes a script name, and loads the contents of the script so it can then be executed.

This method allows IPython magics to be executed as part of a script.

This also useful if you want to run a script, but edit the script before it is run.

```
In [4]: %load imports.py
```

```
In [5]: # %load imports.py
...: import numpy as np
...: import scipy as sp
...: import pandas as pd
...: import matplotlib.pyplot as plt
...: import matplotlib as mpl
...: %matplotlib inline
...: import seaborn as sns
...: sns.set()
...:
...:
```


External commands

- Precede command with !
- Can assign output to variable

Any OS command can be run by starting it with a !.

The resulting output is returned as a list of strings (stripping the trailing \n characters). The result can be assigned to a variable.

Windows

```
In [3]: !dir DATA\*.csv
 Volume in drive Z is Shared Folders
 Volume Serial Number is 0000-0064

 Directory of Z:\Desktop\py2forsci\DATA

02/20/2014  01:53 PM             5,511 airport_boardings.csv
02/20/2014  01:53 PM             2,182 energy_use_quad.csv
02/20/2014  01:53 PM             4,993 parasite_data.csv
               3 File(s)         12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free

In [4]:
```

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv

In [3]:
```


Using history

- use %history magic
- `history` *list commands*
- `history -n` *list commands with numbers*
- `hist` *shortcut for "history"*

The `%history` magic will list previous commands. Use `-n` to list commands with their numbers.

Selecting commands

You can select a single command or a range of commands separated by a dash.

```
history 5
history 6-10
```

Use `~N/`, where N is 1 or greater, to select commands from previous sessions.

```
history ~2/3  third command in second previous session
```

To select more than one range or individual command, separate them by spaces.

```
history 4-6 9 12-16
```

The same syntax can be used with `%edit`, `%rerun`, `%recall`, `%macro`, `%save` and `%pastebin`.

Recalling commands

The `%recall` magic will recall a previous command by number. It will leave the cursor at the end of the command so you can edit it.

```
recall 12
recall 4-7
```

Rerunning commands

`%rerun` will re-run a previous command without waiting for you to press `Enter`.

Saving sessions

- Save commands to Python script
- Specify one or more commands
- Use `%save` magic

It is easy to save a command, a range of commands, or any combination of commands to a Python script using the `%save` magic.

The syntax is

```
%save filename selected commands
```

.py will be appended to the filename.

Using Pastebin

- Online "clipboard"
- Use `%pastebin` command

Pastebin is a free online service that accepts pasted text and provides a link to access the text. It can be used to share code snippets with other programmers.

The `%pastebin` magic will paste selected commands to **Pastebin** and return a link that can be used to retrieve them. The link provided will expire in 7 days.

Use `-d` to specify a title for the pasted code.

```
link = %pastebin -d "my code" 10-15  write commands 10 through 15 to Pastebin and get link
```

Add ".txt" to the link to retrieve the plain text that you pasted. This can be done with `requests`:

```
import requests
link = %pastebin -d "my code" 10-15
pasted_text = requests.get(link + '.txt').text
```


Benchmarking

- Use %timeit

IPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }

In [2]: %timeit red_value = color_values['red']
10000000 loops, best of 3: 54.5 ns per loop

In [3]: %timeit red_value = color_values.get('red')
10000000 loops, best of 3: 115 ns per loop
```

%timeit will benchmark whatever code comes after it on the same line. %%timeit will benchmark contents of a notebook cell

Profiles

- Stored in `.ipython` folder in home folder
- Contains profiles and other configuration
- Can have multiple profiles
- `ipython profile` subcommands
    - `list`
    - `create`
    - `locate`

IPython supports *profiles* for storing custom configurations and startup scripts. There is a default profile, and any number of custom profiles can be created.

Each profile is a separate subfolder under the `.ipython` folder in a users's home folder.

Creating profiles

Use `ipython profile create` `name` to create a new named profile. If `name` is omitted, this will create the default profile (if it does not already exist)

Listing profiles

Use `ipython profile list` to list all profiles

Finding profiles

`ipython profile locate` `name` will display the path to the specified profile. As with creating, omitting the name shows the path to the default profile.


```
.ipython
├── cython
│   └── Users
│       └── mikedev
├── extensions
├── nbextensions
├── profile_default
│   ├── db
│   ├── ipython_config.py
│   ├── ipython_kernel_config.py
│   ├── log
│   ├── pid
│   ├── security
│   ├── startup
│   │   ├── 00_imports.py
│   │   └── 10_macros.py
│   └── static
│       └── custom
└── profile_science
    ├── db
    ├── ipython_config.py
    ├── ipython_kernel_config.py
    ├── log
    ├── pid
    ├── security
    └── startup
        └── 00_imports.py
```

Configuration

IPython has many configuration settings. You can change these settings by creating or editing the script named `ipython_config.py` in a profile folder.

Within this script you can use the global config object, named `c`.

For instance, the line

```
c.InteractiveShellApp.pylab_import_all = False
```

Will change how the `%pylab` magic works. When true, it will populate the user namespace with the contents of `numpy` and `pylab` as though you had entered `from numpy import *` and `from pylab import *`

When false, it will just import `numpy` as `np` and `pylab` as `pylab`.

Link to all IPython options:

https://ipython.readthedocs.io/en/stable/config/options/index.html

There are two groups of settings. "Terminal Python Options" refers to using iPython interactively. "IPython kernal options" refers to using iPython in a Jupyter Notebook.

> When you create a profile, this config script is created with some commented code to get you started.

Startup

Startup scripts allow you to execute frequently used code, especially imports, when starting IPython.

Startup scripts go in the `startup` folder of the profile folder. All Python scripts in this folder will be executed, in lexicographical (sorted) order.

The scripts will be executed in the context of the IPython session, so all imports, variables, functions, classes, and other definitions will be available in the session.

> It is convenient to prefix the startup scripts with "00", "10", "20", and so forth, to set the order of execution.

Jupyter notebooks

- Extension of IPython
- Puts the interpreter in a web browser
- Code is grouped into "cells"
- Cells can be edited, repeated, etc.

In 2015, the developers of IPython pulled the notebook feature out of IPython to make a separate product called Jupyter. It is still invoked via the `jupyter notebook` command, and now supports over 130 language kernels in addition to Python.

A Jupyter notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into cells, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into Jupyter.

JupyterLab

In 2018, the Jupyter developers released **JupyterLab**, which is a web-based application to manage Jupyter notebooks and other files in one place.

JupyterLab Demo

Start JupyterLab and follow along with a demo of JupyterLab and Jupyter notebooks as directed by the instructor

Open an Anaconda prompt (on Windows) or a terminal window (on Mac or Linux) and navigate to the top folder of the student files, then

```
cd NOTEBOOKS
jupyter-lab
```


For more information

- https://ipython.org/
- https://jupyter.org/
- https://ipythonbook.com

Chapter 3: 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

```python
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 2. 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 0.00622 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 0.00150 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 3 Exercises

Exercise 3-1 (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 3-2 (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 3-3 (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 4: 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, Matplotlib, and other scientific packages.

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

pandas provides powerful split-apply-combine operations — **groupby** enables transformations, aggregations, 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 structures: Series and DataFrame
- 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 an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column indexes (like the rows and columns of a spreadsheet, but more flexible).

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

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

Series

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

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

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

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

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

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

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

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

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

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

Example

pandas_series.py

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

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

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

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

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

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

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

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

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

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

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

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

s1['e'] *= 5

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

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

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

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

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

pandas_series.py

```
s1: a    0.594809
b    0.374595
c    0.575765
d   -0.237595
e    0.458586
f   -0.341158
g    0.448783
h    1.831066
i    0.546472
j    0.257281
dtype: float64

s2: 0    0.594809
1    0.374595
2    0.575765
3   -0.237595
4    0.458586
5   -0.341158
6    0.448783
7    1.831066
8    0.546472
9    0.257281
dtype: float64

selecting elements
h    1.831066
b    0.374595
dtype: float64

a    0.594809
b    0.374595
c    0.575765
dtype: float64

slice of elements
b    0.374595
c    0.575765
d   -0.237595
dtype: float64

sum(), mean(), min(), max():
4.5086031775321995 0.45086031775321994 -0.3411578039897354 1.8310657812984552

cumsum(), cumprod():
a    0.594809
```



```
b    0.969404
c    1.545169
d    1.307574
e    1.766160
f    1.425002
g    1.873785
h    3.704850
i    4.251322
j    4.508603
dtype: float64 a    0.594809
b    0.222813
c    0.128288
d   -0.030480
e   -0.013978
f    0.004769
g    0.002140
h    0.003919
i    0.002141
j    0.000551
dtype: float64

True
False

s3 (which is s1 * 10)
a     5.948090
b     3.745952
c     5.757647
d    -2.375946
e     4.585856
f    -3.411578
g     4.487826
h    18.310658
i     5.464720
j     2.572807
dtype: float64

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

```
dtype: bool

assign -1 where mask is true
a     5.948090
b    -1.000000
c     5.757647
d    -1.000000
e    -1.000000
f    -1.000000
g    -1.000000
h    18.310658
i     5.464720
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 create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

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

Table 3. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are union-ed together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are union-ed to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

Most, if not all, of the time you will create Series and Dataframes by reading data.

Example

pandas_simple_dataframe.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column names
indices = ['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('cols')
print(cols, '\n')

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

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

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

print_header("df['gamma']")
print(df['gamma'])  # select column 'gamma'
```

pandas_simple_dataframe.py

```
==================================================
=                      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]]

==================================================
=                  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['gamma']                   =
==================================================
a    120
b    220
c    320
d    420
e    520
f    620
Name: gamma, dtype: int64
```


Reading Data

- Supports many data formats
- Reads headings to create column indexes
- Auto-creates indexes as needed
- Can used specified column as row index

Pandas supports many different input formats. It will read file headings and use them to create column indexes. By default, it will use integers for row indexes, but you can specify a column to use as the index, or provide a list of index values.

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 the most frequently used function, and has many options. It can also be used to read generic flat-file formats. **read_table** is similar to **read_csv()**, but doesn't assume CSV format.

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

See **Jupyter** notebook **pandas_Input_Demo** (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 4. 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()` *basic statistical details*
- `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 `Categorical` 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
```

Basic Indexing

- Similar to normal Python or numpy
- 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, like normal Python, or extended subscripting, similar to numpy. In addition, pandas has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) 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.

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
```

Saner indexing with .loc, .iloc, and .at

- `loc[row-spec,col-spec]` for names (strings or numbers)
- `.iloc[row-spec,col-spec]` for 0-based position (integers only)
- `.loc[]` row or column specs can be
  - single name
  - iterable of names
  - range (inclusive) of names
- `.iloc[]` row or column specs can be
  - single number
  - iterable of numbers
  - range (exclusive) of numbers
- `.at[]` single value

The `.loc` and `.iloc` indexers provide more extensive and consistent selecting of rows and columns for dataframes. They both work exactly the same way, but `.loc` uses only row and column *names*, and `.iloc` uses only *positions*.

Both indexers use the *getitem* operator `[]`, with the syntax `[row-specifier, column-specifier]`.

For `.loc[]`, the specifier can be either a single name, an iterable of names, or a range of names. The end of a range is inclusive.

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

To select all rows, or all columns, use `:`.

The `.at[]` property can be used to select a single value at a given row and column: `df.at[47, "color"]`. This is a shortcut for `.loc[row, col]`.

For `.loc()` and `.iloc()`, the column specifier can be omitted, which will select all columns for those rows.

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
```


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")
```

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
```

Broadcasting

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

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

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

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

print(row_labels, "\n")

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, row_labels, column_labels)  # create dataframe from data
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
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()
```


pandas_broadcasting.py

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

==================================================
=                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 each value                =
==================================================
            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
```

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',
sheet_name='Master',
                  na_values='NA()')
df.index = range(1,len(df)+1)

print(df.head())
print(df.loc[1])
party_counts = df['Political Party'].value_counts()
print(party_counts)
# plot the data
plt.figure(figsize=(20.0,8.0))
party_counts.plot(kind='barh')
plt.show()
```


Creating new columns

- Assign to column with new name
- Use normal operators with other columns

For simple cases, it's easy to create new columns. Just assign a Series-like object to a new column name. The easy way to do this is to combine other columns with an operator or function.

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

print(df)
```

pandas_new_columns.py

```
   alpha  beta  gamma  delta  epsilon    zeta   eta   theta      iota  kappa
a    100   110    120    130      140   18200  1000   19800    4950.0  120.0
b    200   210    220    230      240   55200  2000   58300   14575.0  220.0
c    300   310    320    330      340  112200  3000  116800   29200.0  320.0
d    400   410    420    430      440  189200  4000  195300   48825.0  420.0
e    500   510    520    530      540  286200  5000  293800   73450.0  520.0
f    600   610    620    630      640  403200  6000  412300  103075.0  620.0
```

Removing entries

- Remove rows or columns
- Use drop() method

To remove columns or rows, use the `drop()` method, with the appropriate labels. Use `axis=1` to drop columns, or axis=0 to drop rows.

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

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)
df.drop(['b', 'c'], inplace=True)
print(df)
```


pandas_drop.py

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

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


Useful pandas methods

Table 5. 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 6. 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.

Even 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 will explain how to start Jupyterlab server.

Chapter 4 Exercises

Exercise 4-1 (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 4-2 (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 5: 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 5 Exercises

Exercise 5-1 (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","1995","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006","2007","2008","2009","2010","2011"]
```

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

Index

@

`%history`, 49
`%load`, 47
`%lsmagic`, 45
`%pastebin`, 52
`%recall`, 50
`%rerun`, 50
`%run`, 46
`%save`, 51
%timeit, 53
@pytest.mark.*mark*, 24

A

`arange()`, 67
`array.flat`, 97
assert, 4
assertions, 3

C

conftest.py, 18

D

data types, 103
DataFrame, 116
DataFrame, 123
Django, 18
dtype, 103

E

`empty()`, 63
exception, 7

F

fixtures, 2
fixtures, 10
`full()`, 63

H

hooks, 18

I

In-place operators, 69
IPython
    %timeit, 53
    benchmarking, 53
    configuration, 56
    getting help, 42
    magic commands, 45
    profiles, 54
    quick reference, 42
    recalling commands, 50
    rerunning commands, 50
    saving commands to a file, 51
    selecting commands, 49
    startup, 57
    tab completion, 44
    using %history, 49
Iterating, 97

J

Java, 3
Jupyter notebook, 58

L

`linspace()`, 67
`lsmagic`, 45

M

markers, 3
matplotlib.pyplot, 167
MatPlotLibExamples.ipynb, 169
mock object, 31
mock object, 31

N

ndarray
    iterating, 97
node ID, 26
NumPy
    getting help, 96
`numpy.info()`, 96
`numpy.lookfor()`, 96

O

`ones()`, 63
OS command, 48

P

pandas, 114
  broadcasting, 149
  DataFrame
    initialize, 123
  Dataframe, 116
  drop(), 158
  I/O functions, 131
  indexing, 136
  read_csv(), 127
  reading data, 127
  selecting, 139
  Series, 116
Panel, 116
parametrizing, 21
plt.plot(), 167
plt.show(, 167
plugins, 18
py.test, 5
pymock, 31
pytest, 3, 4
pytest
  builtin fixtures, 14
  configuring fixtures, 18
  output capture, 5
  special assertions, 7
  user-defined fixtures, 11
  verbose, 5
pytest-django, 37
pytest-mock, 32
pytest-qt, 37
pytest.approx(), 7
pytest.fixture, 11
pytest.raises(), 7

R

read_csv(), 127
read_table, 127
redis, 18
running tests, 27
  by component, 27
  by mark, 27
  by name, 27

S

SciPy, 61
Series, 116
Series, 117
shape, 72

T

test case, 2
test cases, 2
test runner, 3, 5
test runners, 2
tests
  messages, 4
tolerance
  pytest.approx, 7

U

ufuncs, 86
ufuncs, 86
  list, 91
unit test, 2
unit test components, 2
unit tests
  failing, 28
  mock objects, 32
  running, 5
  skipping, 28
unittest.mock, 31
unittest.mock, 32

V

vectorize(), 92
vectorized, 86

X

xfail, 28
XPASS, 28
xUnit, 3

Z

`zeros()`, 63

