**The Python Interpreter**

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time.

The standard interactive Python interpreter can be invoked on the command line with the python command:

$ **python**

Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)

>>> a = 5

>>> print(a)

5

The >>> you see is the *prompt* after which you’ll type code expressions.

To exit the Python interpreter, you can either type **exit()** or press Ctrl-D (works on Linux and macOS only).

Running Python programs is as simple as calling python with a *.py* file as its first argument.

Suppose we had created *hello\_world.py* with these contents:

print("Hello world")

You can run it by executing the following command (the *hello\_world.py* file must be in your current working terminal directory):

$ **python hello\_world.py**

Hello world

**IPython(Enhanced Python Intrepreter)**

While some Python programmers execute all of their Python code in this way,those doing **data analysis** or **scientific computing** make use of IPython, an enhanced

Python interpreter, or Jupyter notebooks, web-based code notebooks originally created within the IPython project.

**Running the IPython Shell**

$ **ipython**

Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57

The default IPython prompt adopts the numbered In [2]: style, compared with the standard >>> prompt.

When you use the **%run command**, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it’s

done:

In [1]: %run hello\_world.py

Hello world

In [2]:

In [1]: a = 5

In [2]: a

Out[2]: 5

**IPython Basics**

**Tab Completion**

One of the major improvements over the standard Python shell is *tab completion*, found in many IDEs or other interactive computing analysis environments. While entering expressions in the shell, pressing the Tab key will search the namespace for any variables (objects,

functions, etc.) matching the characters you have typed so far and show the results in a convenient drop-down menu:

In [1]: an\_apple = 27

In [2]: an\_example = 42

In [3]: an**<Tab>**

an\_apple an\_example any

this functionality can save you many keystrokes.

**Object Introspection**

Using a question mark (?) before or after a variable will display some general information about the object:

In [1]: b = [1, 2, 3]

In [2]: b?

Type: list

String form: [1, 2, 3]

Length: 3

Docstring:Built-**in** mutable sequence.If no argument **is** given, the constructor creates a new empty list.

The argument must be an iterable **if** specified.

In [3]: print?

Docstring:

print(value, ..., sep=' ', end='**\n**', file=sys.stdout, flush=**False**)

Prints the values to a stream, **or** to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Type: builtin\_function\_or\_method.

**Python Language Basics**

**Language Semantics**

The Python language design is distinguished by its emphasis on readability, simplicity,and explicitness.

**Indentation, not braces**

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in

many other languages like C++, Java, and Perl.

Consider a for loop from a sorting algorithm:

**for** x **in** array:

**if** x < pivot:

less.append(x)

**else**:

greater.append(x)

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block.

**Everything is an object**

An important characteristic of the Python language is the consistency of its *object model*.

Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own “box,” which is referred to as a *Python object*.

Each object has an associated *type* (e.g., *integer*, *string*, or *function*) and internal data.

**Comments**

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want

to exclude certain blocks of code without deleting them. One solution is to *comment out* the code:

results = []

**for** line **in** file\_handle:

*# keep the empty lines for now*

*# if len(line) == 0:*

*# continue*

results.append(line.replace("foo", "bar"))

Comments can also occur after a line of executed code. While some programmers

prefer comments to be placed in the line preceding a particular line of code, this can

be useful at times:

print("Reached this line") *# Simple status report*

**Function and object method calls**

We call functions using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:

result = f(x, y, z)

g()

Almost every object in Python has attached functions, known as *methods*, that have access to the object’s internal contents. You can call them using the following syntax:

obj.some\_method(x, y, z)

Functions can take both *positional* and *keyword* arguments:

result = f(a, b, c, d=5, e="foo")

We will look at this in more detail later.

**Variables and argument passing**

When assigning a variable (or *name*) in Python, you are creating a *reference* to the object shown on the righthand side of the equals sign. In practical terms, consider a list of integers:

In [8]: a = [1, 2, 3]

Suppose we assign a to a new variable b:

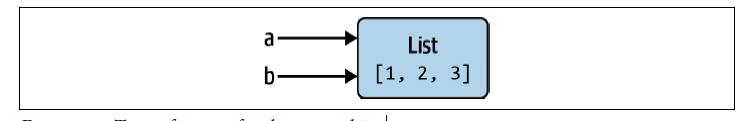
In [9]: b = a

In [10]: b

Out[10]: [1, 2, 3]

In some languages, the assignment if b will cause the data [1, 2, 3] to be copied.

In Python, a and b actually now refer to the same object, the original list [1, 2, 3]



You can prove this to yourself by appending an element to

a and then examining b:

In [11]: a.append(4)

In [12]: b

Out[12]: [1, 2, 3, 4]

**Dynamic references, strong types**

Variables in Python have no inherent type associated with them; a variable can refer to a different type of object simply by doing an assignment. There is no problem with

the following:

In [17]: a = 5

In [18]: type(a)

Out[18]: int

In [19]: a = "foo"

In [20]: type(a)

Out[20]: str

Variables are names for objects within a particular namespace; the type information is stored in the object itself.

Some observers might hastily conclude that Python is not a “typed language.” This is not true; consider this example:

In [21]: "5" + 5

---------------------------------------------------------------------------

**TypeError** Traceback (most recent call last)

<ipython-input-21-7fe5aa79f268> **in** <module>

----> 1 "5" + 5

**TypeError**: can only concatenate str (**not** "int") to str

In Python, such implicit casts are not allowed.

In this regard we say that Python is a *strongly typed* language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in certain permitted circumstances, such as:

In [22]: a = 4.5

In [23]: b = 2

*# String formatting, to be visited later*

In [24]: print(f"a is {type(a)}, b is {type(b)}")

a **is** <**class** '**float**'>, b is <class 'int'>

In [25]: a / b

Out[25]: 2.25

**Attributes and methods**

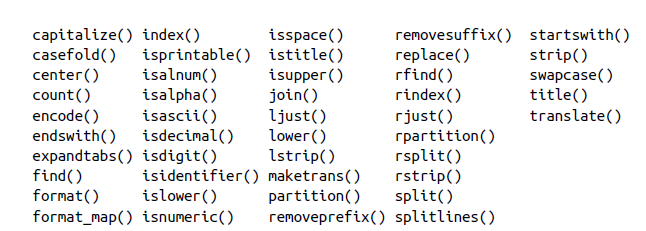
Objects in Python typically have both attributes

and methods. Both of them are accessed via the syntax

*obj.attribute\_name*:

In [1]: a = "foo"

In [2]: a.**<Press Tab>**



**Duck typing**

Often you may not care about the type of an object but rather only whether it has certain methods or behavior.

This is sometimes called *duck typing*, after the saying “If

it walks like a duck and quacks like a duck, then it’s a duck.”

For example, you can verify that an object is iterable if it implements the *iterator protocol*. For many objects,

this means it has an \_\_iter\_\_ “magic method,” though an alternative and better way to check is to try using the iter function:

In [33]: **def** isiterable(obj):

....: **try**:

....: iter(obj)

....: **return True**

....: **except TypeError**: *# not iterable*

....: **return False**

This function would return True for strings as well as most Python collection types:

In [34]: isiterable("a string")

Out[34]: **True**

In [35]: isiterable([1, 2, 3])

Out[35]: **True**

In [36]: isiterable(5)

Out[36]: **False**

**Imports**

In Python, a *module* is simply a file with the *.py* extension containing Python code.

Suppose we had the following module:

*# some\_module.py*

PI = 3.14159

**def** f(x):

**return** x + 2

**def** g(a, b):

**return** a + b

If we wanted to access the variables and functions defined in *some\_module.py*, from another file in the same directory we could do:

**import some\_module**

result = some\_module.f(5)

pi = some\_module.PI

**Binary operators and comparisons**

Most of the binary math operations and comparisons use familiar mathematical syntax used in other programming languages:

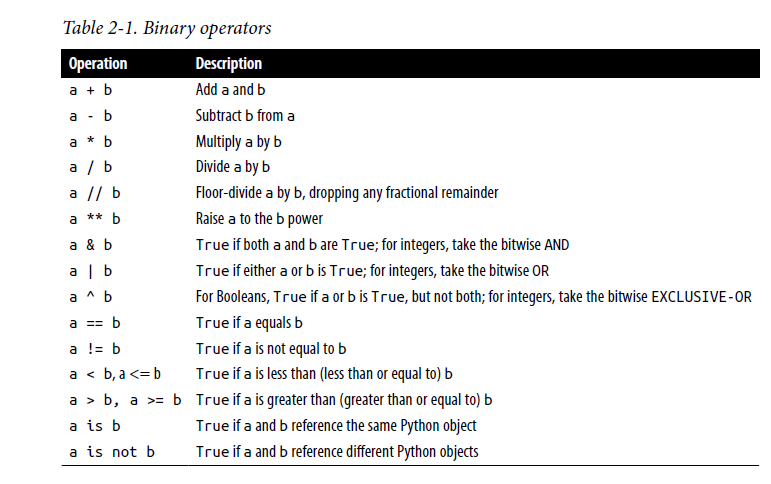
In [37]: 5 - 7

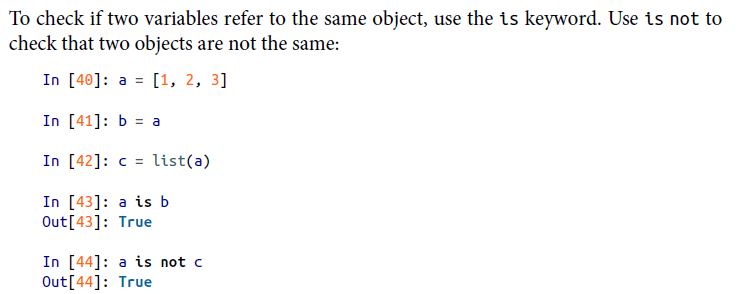
Out[37]: -2

In [38]: 12 + 21.5

Out[38]: 33.5

In [39]: 5 <= 2





**Mutable and immutable objects**

Many objects in Python, such as lists, dictionaries, NumPy arrays, and most userdefined types (classes), are *mutable*. This means that the object or values that they

contain can be modified:

In [48]: a\_list = ["foo", 2, [4, 5]]

In [49]: a\_list[2] = (3, 4)

In [50]: a\_list

Out[50]: ['foo', 2, (3, 4)]

Others, like strings and tuples, are immutable, which means their internal data cannot be changed:

In [51]: a\_tuple = (3, 5, (4, 5))

In [52]: a\_tuple[1] = "four"

---------------------------------------------------------------------------

**TypeError** Traceback (most recent call last)

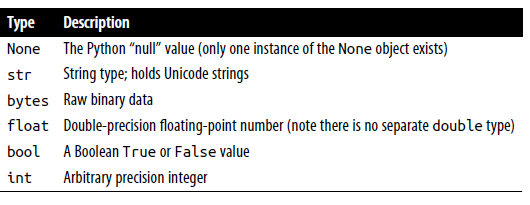
<ipython-input-52-cd2a018a7529> **in** <module>

----> 1 a\_tuple[1] = "four"

**TypeError**: 'tuple' object does **not** support item assignment

**Scalar Types**

Python has a small set of built-in types for handling numerical data, strings, Boolean(True or False) values, and dates and time. These “single value” types are sometimes called *scalar types*, and we refer to them in this book as *scalars* .



**Numeric types**

The primary Python types for numbers are int and float. An int can store arbitrarily

large numbers:

In [53]: ival = 17239871

In [54]: ival \*\* 6

Out[54]: 26254519291092456596965462913230729701102721

Floating-point numbers are represented with the Python float type. Under the hood,each one is a double-precision value. They can also be expressed with scientific notation:

In [55]: fval = 7.243

In [56]: fval2 = 6.78e-5

Integer division not resulting in a whole number will always yield a floating-point

number:

In [57]: 3 / 2

Out[57]: 1.5

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator //:

In [58]: 3 // 2

Out[58]: 1

**Strings**

Many people use Python for its built-in string handling capabilities. You can write *string literals* using either single quotes ' or double quotes " (double quotes are

generally favored):

a = 'one way of writing a string'

b = "another way"

The Python string type is str.

For multiline strings with line breaks, you can use triple quotes, either ''' or """:

c = """

This is a longer string that

spans multiple lines

"""

In [61]: a = "this is a string"

In [63]: b = a.replace("string", "longer string")

In [64]: b

Out[64]: 'this is a longer string'

Afer this operation, the variable a is unmodified:

In [65]: a

Out[65]: 'this is a string'

Many Python objects can be converted to a string using the str function:

In [66]: a = 5.6

In [67]: s = str(a)

In [68]: print(s)

5.6

Strings are a sequence of Unicode characters and therefore can be treated like other

sequences, such as lists and tuples:

In [69]: s = "python"

In [70]: list(s)

Out[70]: ['p', 'y', 't', 'h', 'o', 'n']

In [71]: s[:3]

Out[71]: 'pyt'

String templating or formatting is another important topic.

. String objects have a format method that can be used to substitute formatted arguments into the string, producing a new string:

In [79]: template = "{0:.2f} {1:s} are worth US${2:d}"

In this string:

• {0:.2f} means to format the first argument as a floating-point number with two

decimal places.

• {1:s} means to format the second argument as a string.

• {2:d} means to format the third argument as an exact integer.

To substitute arguments for these format parameters, we pass a sequence of arguments to the format method:

In [80]: template.format(88.46, "Argentine Pesos", 1)

Out[80]: '88.46 Argentine Pesos are worth US$1'

Python 3.6 introduced a new feature called *f-strings* (short for *formatted string literals*) which can make creating formatted strings even more convenient.

To create an fstring, write the character f immediately preceding a string literal.

In [81]: amount = 10

In [82]: rate = 88.46

In [83]: currency = "Pesos"

In [84]: result = f"{amount} {currency} is worth US${amount / rate}"

Format specifiers can be added after each expression using the same syntax as with the string templates above:

In [85]: f"{amount} {currency} is worth US${amount / rate:.2f}"

Out[85]: '10 Pesos is worth US$0.11'

**Bytes and Unicode**

In modern Python (i.e., Python 3.0 and up), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text.

In older versions of Python, strings were all bytes without any explicit Unicode encoding.

You could convert to Unicode assuming you knew the character encoding.

Here is an example Unicode string with non-ASCII characters:

In [86]: val = "español"

In [87]: val

Out[87]: 'español'

We can convert this Unicode string to its UTF-8 bytes representation using the encode method:

In [88]: v = val.encode("utf-8")

In [89]: v

Out[89]: b'espa**\xc3\xb1**ol'

In [90]: type(v)

Out[90]: bytes

In [91]: v.decode("utf-8")

Out[91]: 'español'

**Booleans**

The two Boolean values in Python are written as True and False. Comparisons and other conditional expressions evaluate to either True or False. Boolean values are combined with the and and or keywords:

In [95]: **True and True**

Out[95]: **True**

In [96]: **False or True**

Out[96]: **True**

**Type casting**

The str, bool, int, and float types are also functions that can be used to cast values

to those types:

In [103]: s = "3.14159"

In [104]: fval = float(s)

In [105]: type(fval)

Out[105]: float

In [106]: int(fval)

Out[106]: 3

In [107]: bool(fval)

Out[107]: **True**

In [108]: bool(0)

Out[108]: **False**

Note that most nonzero values when cast to bool become True.

**None**

None is the Python null value type:

In [109]: a = **None**

In [110]: a **is None**

Out[110]: **True**

In [111]: b = 5

In [112]: b **is not None**

Out[112]: **True**

**Dates and times**

The built-in Python datetime module provides datetime, date, and time types.

The datetime type combines the information stored in date and time and is the most commonly used:

In [113]: from datetime import datetime, date, time

In [114]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [115]: dt.day

Out[115]: 29

In [116]: dt.minute

Out[116]: 30

**Control Flow-if,elif,else,for,range,while,range,pass….**

**Built-In Data Structures,Functions, and Files**

**Data Structures and Sequences**

Python’s data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

We start with tuple, list, and dictionary, which are some of the most frequently used sequence types.

**Tuple**

A tuple is a fixed-length, immutable sequence of Python objects which, once assigned, cannot be changed. The easiest way to create one is with a comma-separated sequence of values wrapped in parentheses:

In [2]: tup = (4, 5, 6)

In [3]: tup

Out[3]: (4, 5, 6)

In many contexts, the parentheses can be omitted, so here we could also have written:

In [4]: tup = 4, 5, 6

In [5]: tup

Out[5]: (4, 5, 6)

You can convert any sequence or iterator to a tuple by invoking tuple:

In [6]: tuple([4, 0, 2])

Out[6]: (4, 0, 2)

In [7]: tup = tuple('string')

In [8]: tup

Out[8]: ('s', 't', 'r', 'i', 'n', 'g')

In [10]: nested\_tup = (4, 5, 6), (7, 8)

In [11]: nested\_tup

Out[11]: ((4, 5, 6), (7, 8))

In [12]: nested\_tup[0]

Out[12]: (4, 5, 6)

In [13]: nested\_tup[1]

Out[13]: (7, 8)

**Unpacking tuples**

If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign:

In [20]: tup = (4, 5, 6)

In [21]: a, b, c = tup

In [22]: b

Out[22]: 5

Even sequences with nested tuples can be unpacked:

In [23]: tup = 4, 5, (6, 7)

In [24]: a, b, (c, d) = tup

In [25]: d

Out[25]: 7

Using this functionality you can easily swap variable names, a task that in many languages might look like:

tmp = a

a = b

b = tmp

But, in Python, the swap can be done like this:

In [26]: a, b = 1, 2

In [27]: a

Out[27]: 1

In [28]: b

Out[28]: 2

In [29]: b, a = a, b

There are some situations where you may want to “pluck” a few elements from the beginning of a tuple. There is a special syntax that can do this, \*rest, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

In [34]: values = 1, 2, 3, 4, 5

In [35]: a, b, \*rest = values

In [36]: a

Out[36]: 1

In [37]: b

Out[37]: 2

In [38]: rest

Out[38]: [3, 4, 5]

As a matter of convention, many Python programmers will use

the underscore (\_) for unwanted variables:

In [39]: a, b, \*\_ = values

**Tuple methods**

In [40]: a = (1, 2, 2, 2, 3, 4, 2)

In [41]: a.count(2)

Out[41]: 4

**List**

In contrast with tuples, lists are variable length and their contents can be modified in place. Lists are mutable. You can define them using square brackets [] or using the list type function:

In [42]: a\_list = [2, 3, 7, None]

In [43]: tup = ("foo", "bar", "baz")

In [44]: b\_list = list(tup)

In [45]: b\_list

Out[45]: ['foo', 'bar', 'baz']

In [46]: b\_list[1] = "peekaboo"

In [47]: b\_list

Out[47]: ['foo', 'peekaboo', 'baz']

**Adding and removing elements**

Elements can be appended to the end of the list with the append method:

In [51]: b\_list.append("dwarf")

In [52]: b\_list

Out[52]: ['foo', 'peekaboo', 'baz', 'dwarf']

Using insert you can insert an element at a specific location in the list:

In [53]: b\_list.insert(1, "red")

In [54]: b\_list

Out[54]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']

The insertion index must be between 0 and the length of the list

The inverse operation to insert is pop, which removes and returns an element at a particular index:

In [55]: b\_list.pop(2)

Out[55]: 'peekaboo'

In [56]: b\_list

Out[56]: ['foo', 'red', 'baz', 'dwarf']

Elements can be removed by value with remove, which locates the first such value and removes it from the list:

In [57]: b\_list.append("foo")

In [58]: b\_list

Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [59]: b\_list.remove("foo")

In [60]: b\_list

Out[60]: ['red', 'baz', 'dwarf', 'foo']

**Concatenating and combining lists**

Similar to tuples, adding two lists together with + concatenates them:

In [63]: [4, None, "foo"] + [7, 8, (2, 3)]

Out[63]: [4, None, 'foo', 7, 8, (2, 3)]

If you have a list already defined, you can append multiple elements to it using the extend method:

In [64]: x = [4, None, "foo"]

In [65]: x.extend([7, 8, (2, 3)])

In [66]: x

Out[66]: [4, None, 'foo', 7, 8, (2, 3)]

**Sorting**

You can sort a list in place (without creating a new object) by calling its sort function:

In [67]: a = [7, 2, 5, 1, 3]

In [68]: a.sort()

In [69]: a

Out[69]: [1, 2, 3, 5, 7]

sort has a few options that will occasionally come in handy. One is the ability to pass a secondary sort key—that is, a function that produces a value to use to sort the

objects. For example, we could sort a collection of strings by their lengths:

In [70]: b = ["saw", "small", "He", "foxes", "six"]

In [71]: b.sort(key=len)

In [72]: b

Out[72]: ['He', 'saw', 'six', 'small', 'foxes']

**Slicing**

You can select sections of most sequence types by using slice notation, which in its basic form consists of start:stop passed to the indexing operator []:

In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]

Out[74]: [2, 3, 7, 5]

Slices can also be assigned with a sequence:

In [75]: seq[3:5] = [6, 3]

In [76]: seq

Out[76]: [7, 2, 3, 6, 3, 6, 0, 1]

In [77]: seq[:5]

Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]

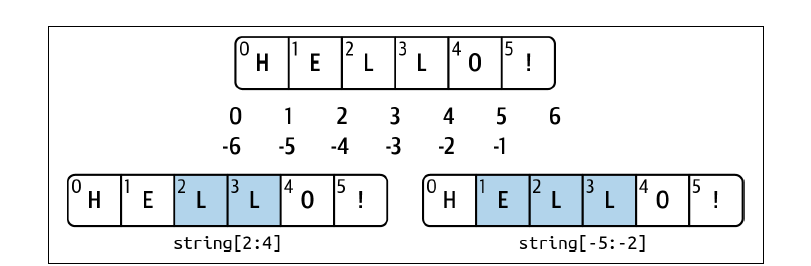
Out[78]: [6, 3, 6, 0, 1]

In [79]: seq[-4:]

Out[79]: [3, 6, 0, 1]

In [80]: seq[-6:-2]

Out[80]: [3, 6, 3, 6]



**Dictionary**

Dictionaries are sometimes called hash maps or

associative arrays. A dictionary stores a collection of key-value pairs, where key and value are Python objects.

In [83]: empty\_dict = {}

In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}

In [85]: d1

Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

In [86]: d1[7] = "an integer"

In [87]: d1

Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [88]: d1["b"]

Out[88]: [1, 2, 3, 4]

You can delete values using either the del keyword or the pop method (which simultaneously returns the value and deletes the key):

In [90]: d1[5] = "some value"

In [91]: d1

Out[91]:

{'a': 'some value',

'b': [1, 2, 3, 4],

7: 'an integer',

5: 'some value'}

In [92]: d1["dummy"] = "another value"

In [93]: d1

Out[93]:{'a': 'some value','b': [1, 2, 3, 4],

7: 'an integer', 5: 'some value','dummy': 'another value'}

In [94]: del d1[5]

In [95]: d1

Out[95]:{'a': 'some value','b': [1, 2, 3, 4],

7: 'an integer','dummy': 'another value'}

In [96]: ret = d1.pop("dummy")

In [97]: ret

Out[97]: 'another value'

In [98]: d1

Out[98]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

The keys and values method gives you iterators of the dictionary’s keys and values, respectively. The order of the keys depends on the order of their insertion, and these functions output the keys and values in the same respective order:

In [99]: list(d1.keys())

Out[99]: ['a', 'b', 7]

In [100]: list(d1.values())

Out[100]: ['some value', [1, 2, 3, 4], 'an integer']

If you need to iterate over both the keys and values, you can use the items method to iterate over the keys and values as 2-tuples:

In [101]: list(d1.items())

Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]

You can merge one dictionary into another using the update method:

In [102]: d1.update({"b": "foo", "c": 12})

In [103]: d1

Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}

**Valid dictionary key types**

While the values of a dictionary can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is hashability. You can check whether an object is hashable (can be used as a key in a dictionary) with the hash function:

In [118]: hash("string")

Out[118]: 3634226001988967898

In [119]: hash((1, 2, (2, 3)))

Out[119]: -9209053662355515447

In [120]: hash((1, 2, [2, 3])) # fails because lists are mutable

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can be:

In [121]: d = {}

In [122]: d[tuple([1, 2, 3])] = 5

In [123]: d

Out[123]: {(1, 2, 3): 5}

**Set**

A set is an unordered collection of unique elements. A set can be created in two ways:

via the set function or via a set literal with curly braces:

In [124]: set([2, 2, 2, 1, 3, 3])

Out[124]: {1, 2, 3}

In [125]: {2, 2, 2, 1, 3, 3}

Out[125]: {1, 2, 3}

Sets support mathematical set operations like union, intersection, difference, and symmetric difference. Consider these two example sets:

In [126]: a = {1, 2, 3, 4, 5}

In [127]: b = {3, 4, 5, 6, 7, 8}

The union of these two sets is the set of distinct elements occurring in either set. This

can be computed with either the union method or the | binary operator:

In [128]: a.union(b)

Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}

In [129]: a | b

Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}

The intersection contains the elements occurring in both sets. The & operator or the intersection method can be used:

In [130]: a.intersection(b)

Out[130]: {3, 4, 5}

In [131]: a & b

Out[131]: {3, 4, 5}

Like dictionary keys, set elements generally must be immutable, and they must be hashable (which means that calling hash on a value does not raise an exception).

**Built-In Sequence Functions**

Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity.

**enumerate**

li=[7, 1, 2, 6, 0, 3, 2]

for x,value in enumerate(li):

print(x)

**Sorted**

The sorted function returns a new sorted list from the elements of any sequence:

In [145]: sorted([7, 1, 2, 6, 0, 3, 2])

Out[145]: [0, 1, 2, 2, 3, 6, 7]

In [146]: sorted("horse race")

Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']

**zip**

zip “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples:

In [147]: seq1 = ["foo", "bar", "baz"]

In [148]: seq2 = ["one", "two", "three"]

In [149]: zipped = zip(seq1, seq2)

In [150]: list(zipped)

Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]

**Reversed**

reversed iterates over the elements of a sequence in reverse order:

In [154]: list(reversed(range(10)))

Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

**List, Set, and Dictionary Comprehensions**

**What are comprehensions?**

1.Comprehensions offer an easy and compact way

Of creating lists, sets and dictionaries.

2.A comprehension works by looping or iterating

Over items and assigning them to a container like

list,set or dictionary.

3.This container cannot be a tuple as tuple being

Immutable is unable to receive assignments.

They take the basic form:

[expr for value in collection if condition]

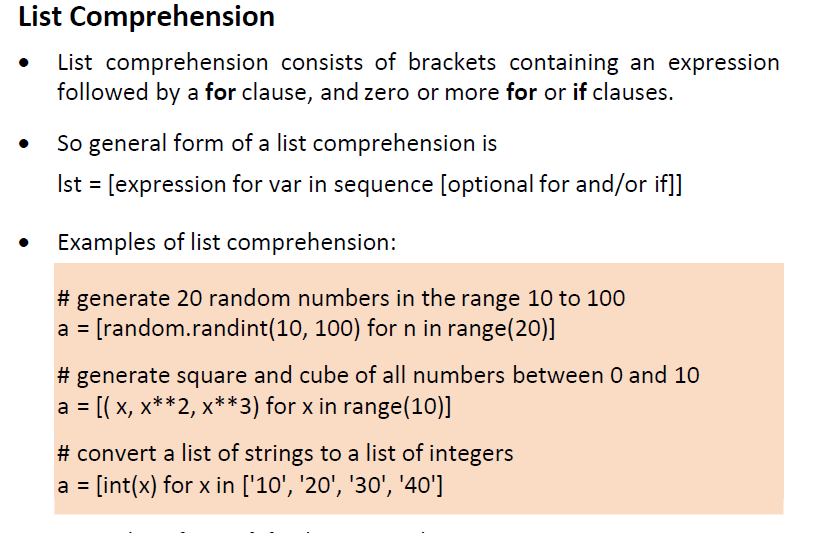
This is equivalent to the following for loop:

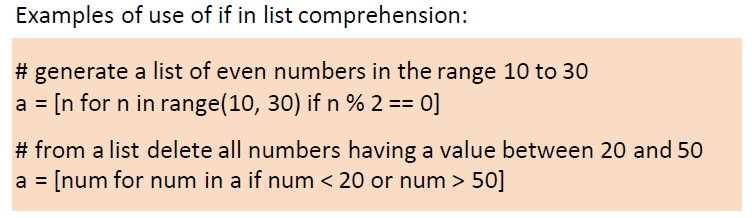
result = []

for value in collection:

if condition:

result.append(expr)





In [155]: strings = ["a", "as", "bat", "car", "dove", "python"]

In [156]: [x.upper() for x in strings if len(x) > 2]

Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON'

**Dictionary Comprehension**

 Dictionary comprehension is a way to create a python dictionary from another dictionary or from any other iterable.

Example :1

# Lists to represent keys and values

keys = ['a','b','c','d','e']

values = [1,2,3,4,5]

myDict = { k:v for (k,v) in zip(keys, values)}

Example :2

myDict = {x: x\*\*2 for x in [1,2,3,4,5]}

print (myDict)

Example :3

customers = ["Alex","Bob","Carol","Dave","Flow","Katie","Nate"]

discount\_dict = {customer:random.randint(1,100) for customer in customers}

print(discount\_dict)

days = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]

temp\_C = [30.5,32.6,31.8,33.4,29.8,30.2,29.9]

weekly\_temp = {day:temp for (day,temp) in zip(days,temp\_C)}

print(weekly\_temp)

**Set Comprehensions**

Set comprehension is a method for creating sets in python using the elements from other iterables like lists, sets, or tuples.

myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

newSet = {element\*3 for element in myList}

print("The existing list is:")

print(myList)

print("The Newly Created set is:")

print(newSet)

Output:

The existing list is:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

The Newly Created set is:

{3, 6, 9, 12, 15, 18, 21, 24, 27, 30}

**Functions**

Functions are the primary and most important method of code organization and reuse in Python.

Example:

In [173]: **def** my\_function(x, y):

.....: **return** x + y

In [174]: my\_function(1, 2)

Out[174]: 3

In [177]: def function\_without\_return(x):

.....: print(x)

In [178]: result = function\_without\_return("hello!")

**Namespaces, Scope, and Local Functions**

def func():

a = []

for i in range(5):

a.append(i)

**Returning Multiple Values**

def f():

a = 5

b = 6

c = 7

return a, b, c

a, b, c = f()

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

x = lambda a : a + 10

print(x(5))

x = lambda a, b : a \* b  
print(x(5, 6))

**Exception handling**

Handling Python errors or exceptions gracefully is an important part of building robust programs.

In [224]: float("1.2345")

Out[224]: 1.2345

In [225]: float("something")

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

Suppose we wanted a version of float that fails gracefully, returning the input

argument.

def attempt\_float(x):

try:

return float(x)

except:

return x

The code in the except part of the block will only be executed if float(x) raises an exception:

In [227]: attempt\_float("1.2345")

Out[227]: 1.2345

In [228]: attempt\_float("something")

Out[228]: 'something'

Example:

try:

a=int(input('Enter an integer:'))

b=int(input('Enter an integer:'))

c=a/b

print('c=',c)

except ZeroDivisionError:

print('Denominator is 0')

You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

def attempt\_float(x):

try:

return float(x)

except (TypeError, ValueError):

return x

**Generators**

In Python, a generator is a [function](https://www.programiz.com/python-programming/function) that returns an [iterator](https://www.programiz.com/python-programming/iterator) that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

Generators in Python are very similar to normal functions with some characteristic differences listed below;

1. Generator functions have yield expression, instead of return used in normal functions.
2. Both yield and return statements return a value from a function. While the return statement ends the function completely, yield statement suspends the function by keeping all its state in the memory for later use.
3. When the generator function yields, the function is not terminated. Instead, yield expression pauses the function and gives control over to the caller.
4. After fully iterated, generators terminate and raise stopIteration exception.

**Example 1:**

**Def mygen():**

**Yield ‘a’**

**Yield ‘b’**

**Yield ‘c’**

**Yield ‘d’**

**g=mygen() // same as l=[‘a’,’b’,’c’,’d’]**

**print(next(g))**

**print(next(g))**

**print(next(g))**

**print(next(g))**

**Example 2:**

**r=[x\*x for x in range(100000000000000000000000000000)]**

print(r)

def sqr()

for x in range(10000000000000000000000000000000)

yield x\*x

res=sqr()

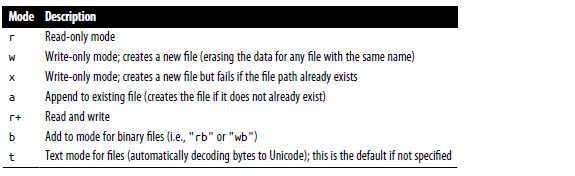
for a in res

print(a)

**Files and the Operating System**

To open a file for reading or writing, use the built-in open function with either a relative or absolute file path and an optional file encoding:

f = open(path, mode,encoding="utf-8")



Here, I pass encoding="utf-8" as a best practice because the default Unicode encoding for reading files varies from platform to platform.

By default, the file is opened in read-only mode "r". We can then treat the file object f like a list and iterate over the lines like so:

**for line in f:**

**print(line)**

When you use open to create file objects, it is recommended to close the file when you are finished with it. Closing the file releases its resources back to the operating system:

**f.close()**

**NumPy ndarray: A Multidimensional Array Object**

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python.

**Creating ndarrays**

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data.

import numpy as np

In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1

Out[21]: array([6. , 7.5, 8. , 0. , 1. ])

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional

array:

In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

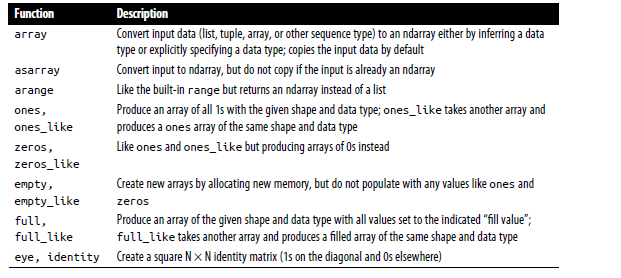
In [23]: arr2 = np.array(data2)

In [24]: arr2

Out[24]:array([[1, 2, 3, 4],

[5, 6, 7, 8]])

**Different funtions used for creating arrays**



**Data Types for ndarrays**

The data type or dtype is a special object containing the information (or metadata,data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

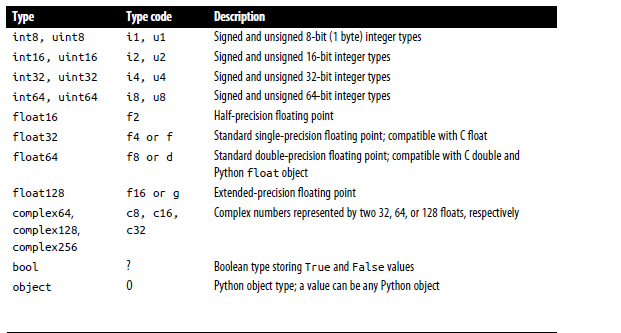
In [35]: arr1.dtype

Out[35]: dtype('float64')

In [36]: arr2.dtype

Out[36]: dtype('int32')

**Numpy Data types**



**Arithmetic with NumPy Arrays**

Arrays are important because they enable you to express batch operations on data without writing any for loops.

NumPy users call this vectorization.

Any arithmetic operations between equal-size arrays apply the operation element-wise:

In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [53]: arr

Out[53]: array([[1., 2., 3.],

[4., 5., 6.]])

In [54]: arr \* arr

Out[54]:

array([[ 1., 4., 9.],

[16., 25., 36.]])

**Basic Indexing and Slicing**

NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements.

One-dimensional arrays are simple; on the surface they act similarly to Python lists:

In [61]: arr = np.arange(10)

In [62]: arr

Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]

Out[63]: 5

In [64]: arr[5:8]

Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr

Out[66]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])

**Fancy Indexing**

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays.

Suppose we had an 8 × 4 array:

In [122]: arr

Out[122]:

array([[0., 0., 0., 0.],

[1., 1., 1., 1.],

[2., 2., 2., 2.],

[3., 3., 3., 3.],

[4., 4., 4., 4.],

[5., 5., 5., 5.],

[6., 6., 6., 6.],

[7., 7., 7., 7.]])

To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

In [123]: arr[[4, 3, 0, 6]]

Out[123]:

array([[4., 4., 4., 4.],

[3., 3., 3., 3.],

[0., 0., 0., 0.],

[6., 6., 6., 6.]])

Hopefully this code did what you expected! Using negative indices selects rows from

the end:

In [124]: arr[[-3, -5, -7]]

Out[124]:

array([[5., 5., 5., 5.],

[3., 3., 3., 3.],

[1., 1., 1., 1.]])

Dimensional array of elements corresponding to each tuple of indices:

In [125]: arr = np.arange(32).reshape((8, 4))

In [126]: arr

Out[126]:

array([[ 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]])

In [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]

Out[127]: array([ 4, 23, 29, 10])

**Transposing Arrays and Swapping Axes**

Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and the special T attribute:

In [132]: arr = np.arange(15).reshape((3, 5))

In [133]: arr

Out[133]:array([[ 0, 1, 2, 3, 4],

[ 5, 6, 7, 8, 9],

[10, 11, 12, 13, 14]])

In [134]: arr.T

Out[134]:

array([[ 0, 5, 10],

[ 1, 6, 11],

[ 2, 7, 12],

[ 3, 8, 13],

[ 4, 9, 14]])Simple transposing with .T is a special case of swapping axes. ndarray has the method swapaxes, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

In [139]: arr

Out[139]:

array([[ 0, 1, 0],

[ 1, 2, -2],

[ 6, 3, 2],

[-1, 0, -1],

[ 1, 0, 1]])

In [140]: arr.swapaxes(0, 1)

Out[140]:array([[ 0, 1, 6, -1, 1],

[ 1, 2, 3, 0, 0],

[ 0, -2, 2, -1, 1]])

**Pseudorandom Number Generation**

The numpy.random module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

For example, you can get a 4 × 4 array of samples from the

standard normal distribution using **numpy.random.standard\_normal**:

In [141]: samples = np.random.standard\_normal(size=(4, 4))

In [142]: samples

Out[142]:

array([[-0.2047, 0.4789, -0.5194, -0.5557],

[ 1.9658, 1.3934, 0.0929, 0.2817],[ 0.769 , 1.2464, 1.0072, -1.2962],

[ 0.275 , 0.2289, 1.3529, 0.8864]])

Python’s built-in random module, by contrast, samples only one value at a time. As you can see from this benchmark, numpy.random is well over an order of magnitude faster for generating very large samples

These random numbers are not truly random (rather, pseudorandom) but instead are generated by a configurable random number generator that determines determin‐

istically what values are created.

Functions like numpy.random.standard\_normal use

the numpy.random module’s default random number generator, but your code can be configured to use an explicit generator:

In [147]: rng = np.random.default\_rng(seed=12345)

In [148]: data = rng.standard\_normal((2, 3))

The seed argument is what determines the initial state of the generator, and the state changes each time the rng object is used to generate data. The generator object rng is

also isolated from other code which might use the numpy.random module:

**Universal Functions: Fast Element-Wise Array Functions**

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays

Many ufuncs are simple element-wise transformations, like **numpy.sqrt or numpy.exp:**

In [150]: arr = np.arange(10)

In [151]: arr

Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [152]: np.sqrt(arr)

Out[152]:array([0. , 1. , 1.4142, 1.7321, 2. , 2.2361, 2.4495, 2.6458,2.8284, 3. ])

In [153]: np.exp(arr)

Out[153]:array([ 1. , 2.7183, 7.3891, 20.0855, 54.5982, 148.4132,403.4288, 1096.6332, 2980.958 , 8103.0839])

These are referred to as unary ufuncs. Others, such as numpy.add or numpy.maximum, take two arrays (thus, binary ufuncs) and return a single array as the result:

**Pandas**

Pandas contains data structures and data manipulation tools designed to make data cleaning and analysis fast and convenient in Python.

Pandas is often used along with libraries like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib.

Pandas adopts NumPy’s idiomatic style of array-based computing without for loops.

Pandas is designed for working with tabular or heterogeneous data. Whereas NumPy, is best suited for working with homogeneously typed numerical array data.

Importing Pandas:

**import pandas as pd**

**Differences Between Pandas VS NumPy is as follows:**

| **PANDAS** | **NUMPY** |
| --- | --- |
| When we have to work on **Tabular data**, we prefer the p*andas*module. | When we have to work on **Numerical data**, we prefer the n*umpy* module. |
| The powerful tools of pandas are**Data frame and Series.** | Whereas the powerful tool of *numpy* is **Arrays.** |
| *Pandas* consume **more** **memory**. | *Numpy* is **memory efficient.** |
| *Pandas*has a better performance when a number of rows is **500K or more.** | *Numpy*has a better performance when number of rows is **50K or less.** |
| Indexing of the *pandas* series is **very slow** as compared to *numpy* arrays. | Indexing of *numpy* Arrays is**very fast**. |
| *Pandas* offer a have2d table object called **DataFrame.** | *Numpy* is capable of providing **multi-dimensional arrays.** |
| It was developed by Wes McKinney and was released in 2008. | It was developed by Travis Oliphant and was released in 2005. |
| It is used in a lot of organizations like Kaidee, Trivago, Abeja Inc. , and a lot more. | It is being used in organizations like Walmart Tokopedia, Instacart, and many more. |
| It has a higher industry application. | It has a lower industry application. |

**Introduction to pandas Data Structures**

The two important datastructures of pandas are Series and Dataframe.

**Series**

A Series is a one-dimensional array-like object containing a sequence of values of the same type and an associated array of data labels,

called its *index*.

The simplest Series is formed from only an array of data:

In [14]: obj = pd.Series([4, 7, -5, 3])

In [15]: obj

Out[15]:

0 4

1 7

2 -5

3 3

dtype: int64

Series consists of two attributes **array** and **index**.

In [16]: obj.array

Out[16]:<PandasArray>

[4, 7, -5, 3]

Length: 4, dtype: int64

In [17]: obj.index

Out[17]: RangeIndex(start=0, stop=4, step=1)

Often, we create a Series with an index identifying each data point with a label:

In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

In [19]: obj2

Out[19]:d 4

b 7

a -5

c 3

dtype: int64

In [20]: obj2.index

Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

In [21]: obj2["a"]

Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]

Out[23]:

c 3

a -5

d 6

In [24]: obj2[obj2 > 0]

Out[24]:d 6

b 7

c 3

dtype: int64

In [25]: obj2 \* 2

Out[25]:d 12

b 14

a -10

c 6

we can convert a dictionary to a pandas series as follows.

In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3

Out[32]:

Ohio 35000

Texas 71000

Oregon 16000

Utah 5000

dtype: int64

A Series can be converted back to a dictionary with its to\_dict method:

In [33]: obj3.to\_dict()

Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)

In [36]: obj4

Out[36]:

California NaN

Ohio 35000.0

Oregon 16000.0

Texas 71000.0

dtype: float64

Here, three values found in sdata were placed in the appropriate locations, but since no value for "California" was found, it appears as NaN (Not a Number), which is considered in pandas to mark missing or NA values.

Since "Utah" was not included in states, it is excluded from the resulting object.

The terms “missing,” “NA,” or “null” are interchangeably to refer to missing data.

The isna and notna functions in pandas should be used to detect missing data:

In [37]: pd.isna(obj4)

Out[37]:

California True

Ohio False

Oregon False

Texas False

dtype: bool

In [38]: pd.notna(obj4)

Out[38]:

California False

Ohio True

Oregon True

Texas True

dtype: bool

Series also has these as instance methods:

In [39]: obj4.isna()

Out[39]:

California True

Ohio False

Oregon False

Texas False

dtype: bool

**DataFrame**

A DataFrame represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different value type (numeric, string,Boolean, etc.).

The DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index.

There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:

data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],

"year": [2000, 2001, 2002, 2001, 2002, 2003],

"pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame = pd.DataFrame(data)

The resulting DataFrame will have its index assigned automatically, as with Series,

and the columns are placed according to the order of the keys in data (which

depends on their insertion order in the dictionary):

In [50]: frame

Out[50]:

state year pop

0 Ohio 2000 1.5

1 Ohio 2001 1.7

2 Ohio 2002 3.6

3 Nevada 2001 2.4

4 Nevada 2002 2.9

5 Nevada 2003 3.2

For large DataFrames, the head method selects only the first five rows:

In [51]: frame.head()

Out[51]:

state year pop

0 Ohio 2000 1.5

1 Ohio 2001 1.7

2 Ohio 2002 3.6

3 Nevada 2001 2.4

4 Nevada 2002 2.9

Similarly, tail returns the last five rows:

In [52]: frame.tail()

Out[52]:

state year pop

1 Ohio 2001 1.7

2 Ohio 2002 3.6

3 Nevada 2001 2.4

4 Nevada 2002 2.9

5 Nevada 2003 3.2

If you specify a sequence of columns, the DataFrame’s columns will be arranged in that order:

In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])

Out[53]:

year state pop

0 2000 Ohio 1.5

1 2001 Ohio 1.7

2 2002 Ohio 3.6

3 2001 Nevada 2.4

4 2002 Nevada 2.9

5 2003 Nevada 3.2

data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],

"year": [2000, 2001, 2002, 2001, 2002, 2003],

"pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

If you pass a column that isn’t contained in the dictionary, it will appear with missing values in the result:

In [54]: frame2 = pd.DataFrame(data,columns=["year", "state", "pop", "debt"])

In [55]: frame2

Out[55]:

year state pop debt

0 2000 Ohio 1.5 NaN

1 2001 Ohio 1.7 NaN

2 2002 Ohio 3.6 NaN

3 2001 Nevada 2.4 NaN

4 2002 Nevada 2.9 NaN

5 2003 Nevada 3.2 NaN

A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by using the dot attribute notation:

In [57]: frame2["state"]

Out[57]:

0 Ohio

1 Ohio

2 Ohio

3 Nevada

4 Nevada

5 Nevada

Name: state, dtype: object

In [58]: frame2.year

Out[58]:

0 2000

1 2001

2 2002

3 2001

4 2002

5 2003

Name: year, dtype: int64

Rows can also be retrieved by position or name with the special iloc and loc attributes

In [59]: frame2.loc[1]

Out[59]:

year 2001

state Ohio

pop 1.7

debt NaN

Name: 1, dtype: object

In [60]: frame2.iloc[2]

Out[60]:

year 2002

state Ohio

pop 3.6

debt NaN

Name: 2, dtype: object

The del method can then be used to remove this column:

In [70]: del frame2["eastern"]

In [71]: frame2.columns

Out[71]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

Another common form of data is a nested dictionary of dictionaries:

In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},

"Nevada": {2001: 2.4, 2002: 2.9}}

If the nested dictionary is passed to the dataFrame, pandas will interpret the outer dictionary keys as the columns, and the inner keys as the row indices:

In [73]: frame3 = pd.DataFrame(populations)

In [74]: frame3

Out[74]:

Ohio Nevada

2000 1.5 NaN

2001 1.7 2.4

2002 3.6 2.9

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

In [75]: frame3.T

Out[75]:

2000 2001 2002

Ohio 1.5 1.7 3.6

Nevada NaN 2.4 2.9

If a DataFrame’s index and columns have their name attributes set, these will also be displayed:

In [79]: frame3.index.name = "year"

In [80]: frame3.columns.name = "state"

In [81]: frame3

Out[81]:

state Ohio Nevada

year

2000 1.5 NaN

2001 1.7 2.4

2002 3.6 2.9

Unlike Series, DataFrame does not have a name attribute. DataFrame’s to\_numpy

method returns the data contained in the DataFrame as a two-dimensional ndarray:

In [82]: frame3.to\_numpy()

Out[82]:

array([[1.5, nan],

[1.7, 2.4],

[3.6, 2.9]])

**Index Objects**

pandas’s Index objects are responsible for holding the axis labels (including a Data‐Frame’s column names) and other metadata (like the axis name or names).

Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

In [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])

In [85]: index = obj.index

In [86]: index

Out[86]: Index(['a', 'b', 'c'], dtype='object')

In [87]: index[1:]

Out[87]: Index(['b', 'c'], dtype='object')

Index objects are immutable and thus can’t be modified by the user:

index[1] = "d" # TypeError

Immutability makes it safer to share Index objects among data structures:

In [88]: labels = pd.Index(np.arange(3))

In [89]: labels

Out[89]: Int64Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [91]: obj2

Out[91]:

0 1.5

1 -2.5

2 0.0

dtype: float64

In [92]: obj2.index is labels

Out[92]: True

In addition to being array-like, an Index also behaves like a fixed-size set:

In [93]: frame3

Out[93]:

state Ohio Nevada

year

2000 1.5 NaN

2001 1.7 2.4

2002 3.6 2.9

In [94]: frame3.columns

Out[94]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [95]: "Ohio" in frame3.columns

Out[95]: True

In [96]: 2003 in frame3.index

Out[96]: False

Unlike Python sets, a pandas Index can contain duplicate labels:

In [97]: pd.Index(["foo", "foo", "bar", "bar"])

Out[97]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')

**Essential Functionality**

**Reindexing**

An important method on pandas objects is reindex, which means to create a new

object with the values rearranged to align with the new index. Consider an example:

In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])

In [99]: obj

Out[99]:

d 4.5

b 7.2

a -5.3

c 3.6

dtype: float64

Calling reindex on this Series rearranges the data according to the new index,introducing missing values if any index values were not already present:

In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])

In [101]: obj2

Out[101]:

a -5.3

b 7.2

c 3.6

d 4.5

e NaN

dtype: float64

In [102]: obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

In [103]: obj3

Out[103]:

0 blue

2 purple

4 yellow

dtype: object

In [104]: obj3.reindex(np.arange(6), method="ffill")

Out[104]:

0 blue

1 blue

2 purple

3 purple

4 yellow

5 yellow

dtype: object

With DataFrame, reindex can alter the (row) index, columns, or both. When passed

only a sequence, it reindexes the rows in the result:

In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),

.....: index=["a", "c", "d"],

.....: columns=["Ohio", "Texas", "California"])

In [106]: frame

Out[106]:

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])

In [108]: frame2

Out[108]:

Ohio Texas California

a 0.0 1.0 2.0

b NaN NaN NaN

c 3.0 4.0 5.0

d 6.0 7.0 8.0

The columns can be reindexed with the columns keyword:

In [109]: states = ["Texas", "Utah", "California"]

In [110]: frame.reindex(columns=states)

Out[110]:

Texas Utah California

a 1 NaN 2

c 4 NaN 5

d 7 NaN 8

Because "Ohio" was not in states, the data for that column is dropped from the result.

Another way to reindex a particular axis is to pass the new axis labels as a positional argument and then specify the axis to reindex with the axis keyword:

In [111]: frame.reindex(states, axis="columns")

Out[111]:

Texas Utah California

a 1 NaN 2

c 4 NaN 5

d 7 NaN 8

**Dropping Entries from an Axis**

In [113]: obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])

In [114]: obj

Out[114]:

a 0.0

b 1.0

c 2.0

d 3.0

e 4.0

dtype: float64

In [115]: new\_obj = obj.drop("c")

In [116]: new\_obj

Out[116]:

a 0.0

b 1.0

d 3.0

e 4.0

dtype: float64

With DataFrame, index values can be deleted from either axis. To illustrate this, we

first create an example DataFrame:

In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),

index=["Ohio", "Colorado", "Utah", "New York"],

columns=["one", "two", "three", "four"])

In [119]: data

Out[119]:

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

In [120]: data.drop(index=["Colorado", "Ohio"])

Out[120]:

one two three four

Utah 8 9 10 11

New York 12 13 14 15

To drop labels from the columns, instead use the columns keyword:

In [121]: data.drop(columns=["two"])

Out[121]:

one three four

Ohio 0 2 3

Colorado 4 6 7

Utah 8 10 11

New York 12 14 15

You can also drop values from the columns by passing axis=1 (which is like NumPy)

or axis="columns":

In [122]: data.drop("two", axis=1)

Out[122]:

one three four

Ohio 0 2 3

Colorado 4 6 7

Utah 8 10 11

New York 12 14 15

In [123]: data.drop(["two", "four"], axis="columns")

Out[123]:

one three

Ohio 0 2

Colorado 4 6

Utah 8 10

New York 12 14

**Indexing, Selection, and Filtering**

In [124]: obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])

In [125]: obj

Out[125]:

a 0.0

b 1.0

c 2.0

d 3.0

dtype: float64

In [126]: obj["b"]

Out[126]: 1.0

In [127]: obj[1]

Out[127]: 1.0

In [128]: obj[2:4]

Out[128]:

c 2.0

d 3.0

dtype: float64

In [129]: obj[["b", "a", "d"]]

Out[129]:

b 1.0

a 0.0

d 3.0

dtype: float64

In [130]: obj[[1, 3]]

Out[130]:

b 1.0

d 3.0

dtype: float64

In [131]: obj[obj < 2]

Out[131]:

a 0.0

b 1.0

dtype: float64

The preferred way to select index values is with the special loc operator:

In [132]: obj.loc[["b", "a", "d"]]

Out[132]:

b 1.0

a 0.0

d 3.0

dtype: float64

In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [135]: obj1

Out[135]: 2 1

0 2

1 3

dtype: int64

In [136]: obj2

Out[136]:

a 1

b 2

c 3

dtype: int64

In [137]: obj1[[0, 1, 2]]

Out[137]:

0 2

1 3

2 1

dtype: int64

In [138]: obj2[[0, 1, 2]]

Out[138]:

a 1

b 2

c 3

dtype: int64

When using loc, the expression obj.loc[[0, 1, 2]] will fail when the index does not contain integers:

In [134]: obj2.loc[[0, 1]]

---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

/tmp/ipykernel\_804589/4185657903.py in <module>

----> 1 obj2.loc[[0, 1]]

^ LONG EXCEPTION ABBREVIATED ^

KeyError: "None of [Int64Index([0, 1], dtype="int64")] are in the [index]"

there is also an iloc operator

that indexes exclusively with integers to work consistently whether or not the index

contains integers:

In [139]: obj1.iloc[[0, 1, 2]]

Out[139]:

2 1

0 2

1 3

dtype: int64

In [140]: obj2.iloc[[0, 1, 2]]

Out[140]:

a 1

b 2

c 3

dtype: int64

Assigning values using these methods modifies the corresponding section of the Series:

In [142]: obj2.loc["b":"c"] = 5

In [143]: obj2

Out[143]:

a 1

b 5

c 5

dtype: int64

In [144]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),

.....: index=["Ohio", "Colorado", "Utah", "New York"],

.....: columns=["one", "two", "three", "four"])

In [145]: data

Out[145]:

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

In [146]: data["two"]

Out[146]:

Ohio 1

Colorado 5

Utah 9

New York 13

Name: two, dtype: int64

In [147]: data[["three", "one"]]

Out[147]:

three one

Ohio 2 0

Colorado 6 4

Utah 10 8

New York 14 12

Another use case is indexing with a Boolean DataFrame, such as one produced by a scalar comparison. Consider a DataFrame with all Boolean values produced by

comparing with a scalar value:

In [150]: data < 5

Out[150]:

one two three four

Ohio True True True True

Colorado True False False False

Utah False False False False

New York False False False False

In [151]: data[data < 5] = 0

In [152]: data

Out[152]:

one two three four

Ohio 0 0 0 0

Colorado 0 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

**Arithmetic and Data Alignment of series and Dataframes**

In [182]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a", "c", "d", "e"])

In [183]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],

.....: index=["a", "c", "e", "f", "g"])

In [184]: s1

Out[184]:

a 7.3

c -2.5

d 3.4

e 1.5

dtype: float64

In [185]: s2

Out[185]:

a -2.1

c 3.6

e -1.5

f 4.0

g 3.1

dtype: float64

Adding these yields:

In [186]: s1 + s2

Out[186]:

a 5.2

c 1.1

d NaN

e 0.0

f NaN

g NaN

dtype: float64

In the case of DataFrame, alignment is performed on both rows and columns:

In [187]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),index=["Ohio", "Texas", "Colorado"])

In [188]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),index=["Utah", "Ohio", "Texas", "Oregon"])

In [189]: df1

Out[189]:

b c d

Ohio 0.0 1.0 2.0

Texas 3.0 4.0 5.0

Colorado 6.0 7.0 8.0

In [190]: df2

Out[190]:

b d e

Utah 0.0 1.0 2.0

Ohio 3.0 4.0 5.0

Texas 6.0 7.0 8.0

Oregon 9.0 10.0 11.0

Adding these returns a DataFrame with index and columns that are the unions of the

ones in each DataFrame:

In [191]: df1 + df2

Out[191]:

b c d e

Colorado NaN NaN NaN NaN

Ohio 3.0 NaN 6.0 NaN

Oregon NaN NaN NaN NaN

Texas 9.0 NaN 12.0 NaN

Utah NaN NaN NaN NaN

**Sorting and Ranking**

Sorting a dataset by some criterion is another important built-in operation. To sort

lexicographically by row or column label, use the sort\_index method, which returns

a new, sorted object:

In [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])

In [235]: obj

Out[235]:

d 0

a 1

b 2

c 3

dtype: int64

In [236]: obj.sort\_index()

Out[236]:

a 1

b 2

c 3

d 0

dtype: int64

With a DataFrame, you can sort by index on either axis:

In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),

index=["three", "one"],columns=["d", "a", "b", "c"])

In [238]: frame

Out[238]:

d a b c

three 0 1 2 3

one 4 5 6 7

In [239]: frame.sort\_index()

Out[239]:

d a b c

one 4 5 6 7

three 0 1 2 3

In [240]: frame.sort\_index(axis="columns")

Out[240]:

a b c d

three 1 2 3 0

one 5 6 7 4

The data is sorted in ascending order by default but can be sorted in descending order, too:

In [241]: frame.sort\_index(axis="columns", ascending=False)

Out[241]:

d c b a

three 0 3 2 1

one 4 7 6 5

To sort a Series by its values, use its sort\_values method:

In [242]: obj = pd.Series([4, 7, -3, 2])

In [243]: obj.sort\_values()

Out[243]:

2 -3

3 2

0 4

1 7

dtype: int64

Any missing values are sorted to the end of the Series by default:

In [244]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [245]: obj.sort\_values()

Out[245]:

4 -3.0

5 2.0

0 4.0

2 7.0

1 NaN

3 NaN

dtype: float64

**Ranking** assigns ranks from one through the number of valid data points in an array, starting from the lowest value.

In [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [252]: obj.rank()

Out[252]:

0 6.5

1 1.0

2 6.5

3 4.5

4 3.0

5 2.0

6 4.5

dtype: float64

DataFrame can compute ranks over the rows or the columns:

In [255]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1], "c": [-2, 5, 8, -2.5]})

In [256]: frame

Out[256]:

b a c

0 4.3 0 -2.0

1 7.0 1 5.0

2 -3.0 0 8.0

3 2.0 1 -2.5

In [257]: frame.rank(axis="columns")

Out[257]:

b a c

0 3.0 2.0 1.0

1 3.0 1.0 2.0

2 1.0 2.0 3.0

3 3.0 2.0 1.0

**Axis Indexes with Duplicate Labels**

In [258]: obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])

In [259]: obj

Out[259]:

a 0

a 1

b 2

b 3

c 4

dtype: int64

The is\_unique property of the index can tell you whether or not its labels are unique:

In [260]: obj.index.is\_unique

Out[260]: False

Data selection is one of the main things that behaves differently with duplicates.

Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

In [261]: obj["a"]

Out[261]:

a 0

a 1

dtype: int64

In [262]: obj["c"]

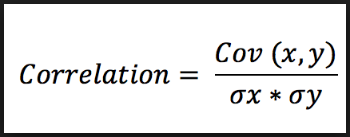
Out[262]: 4

**Correlation and Covariance**

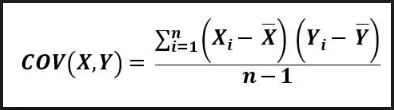
Correlation, statistical technique which determines how one variables moves/changes in relation with the other variable.

It gives us the idea about the degree of the relationship of the two variables.

It’s a bi-variate analysis measure which describes the association between different variables.



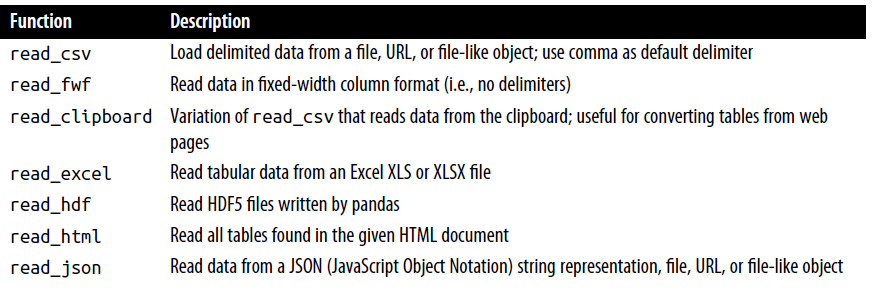
Covariance: The prefix ‘Co’ defines some kind of joint action and variance refers to the change or variation. So it says, two variables are related based on how these variables change in relation with each other.

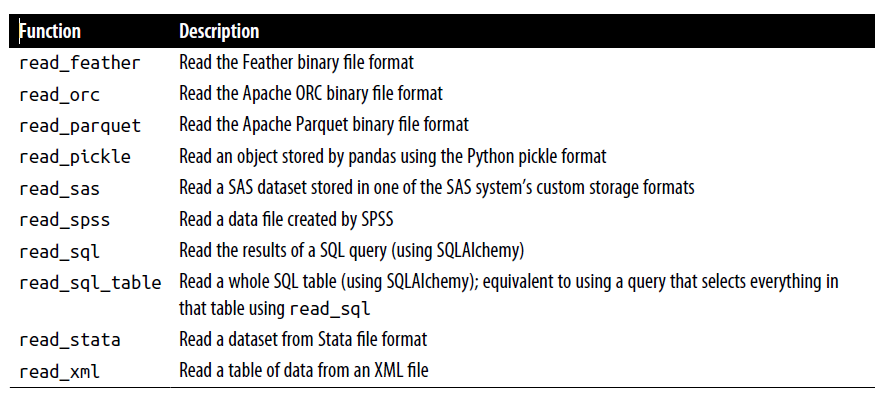


**Reading and Writing Data in Text Format**

In Pandas there are several functions used for reading data into a dataframe.

pandas.read\_csv is one of the most frequently used fuction.





Let’s start with a small comma-separated values (CSV) text file:

In [10]: !cat examples/ex1.csv

a,b,c,d,message

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo

Since this is comma-delimited, we can then use pandas.read\_csv to read it into a DataFrame:

In [11]: df = pd.read\_csv("examples/ex1.csv")

In [12]: df

Out[12]:

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

A file will not always have a header row. Consider this file:

In [13]: !cat examples/ex2.csv

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

In [14]: pd.read\_csv("examples/ex2.csv", header=None)

Out[14]:

0 1 2 3 4

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

In [15]: pd.read\_csv("examples/ex2.csv", names=["a", "b", "c", "d", "message"])

Out[15]: a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

If you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

In [18]: !cat examples/csv\_mindex.csv

key1,key2,value1,value2

one,a,1,2

one,b,3,4

one,c,5,6

one,d,7,8

two,a,9,10

two,b,11,12

two,c,13,14

two,d,15,16

In [19]: parsed = pd.read\_csv("examples/csv\_mindex.csv",

....: index\_col=["key1", "key2"])

In [20]: parsed

Out[20]:

value1 value2

key1 key2

one a 1 2

b 3 4

c 5 6

d 7 8

two a 9 10

b 11 12

c 13 14

d 15 16

In some cases, a table might not have a fixed delimiter, using whitespace or some

other pattern to separate fields. Consider a text file that looks like this:

In [21]: !cat examples/ex3.txt

A B C

aaa -0.264438 -1.026059 -0.619500

bbb 0.927272 0.302904 -0.032399

ccc -0.264273 -0.386314 -0.217601

ddd -0.871858 -0.348382 1.100491

In [22]: result = pd.read\_csv("examples/ex3.txt", sep="\s+")

In [23]: result

Out[23]:

A B C

aaa -0.264438 -1.026059 -0.619500

bbb 0.927272 0.302904 -0.032399

ccc -0.264273 -0.386314 -0.217601

ddd -0.871858 -0.348382 1.100491

In [24]: !cat examples/ex4.csv

# hey!

a,b,c,d,message

# just wanted to make things more difficult for you

# who reads CSV files with computers, anyway?

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo

In [25]: pd.read\_csv("examples/ex4.csv", skiprows=[0, 2, 3])

Out[25]:

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

**Reading Text Files in Pieces**

Before we look at a large file, we make the pandas display settings more compact:

In [40]: pd.options.display.max\_rows = 10

Now we have:

In [41]: result = pd.read\_csv("examples/ex6.csv")

In [42]: result

Out[42]:

one two three four key

0 0.467976 -0.038649 -0.295344 -1.824726 L

1 -0.358893 1.404453 0.704965 -0.200638 B

2 -0.501840 0.659254 -0.421691 -0.057688 G

3 0.204886 1.074134 1.388361 -0.982404 R

4 0.354628 -0.133116 0.283763 -0.837063 Q

... ... ... ... ... ..

9995 2.311896 -0.417070 -1.409599 -0.515821 L

9996 -0.479893 -0.650419 0.745152 -0.646038 E

9997 0.523331 0.787112 0.486066 1.093156 K

9998 -0.362559 0.598894 -1.843201 0.887292 G

9999 -0.096376 -1.012999 -0.657431 -0.573315 0

[10000 rows x 5 columns]

If you want to read only a small number of rows (avoiding reading the entire file),specify that with nrows:

In [43]: pd.read\_csv("examples/ex6.csv", nrows=5)

Out[43]:

one two three four key

0 0.467976 -0.038649 -0.295344 -1.824726 L

1 -0.358893 1.404453 0.704965 -0.200638 B

2 -0.501840 0.659254 -0.421691 -0.057688 G

3 0.204886 1.074134 1.388361 -0.982404 R

4 0.354628 -0.133116 0.283763 -0.837063 Q

**Writing Data to Text Format**

Data can also be exported to a delimited format. Let’s consider one of the CSV files read before:

In [48]: data = pd.read\_csv("examples/ex5.csv")

In [49]: data

Out[49]:

something a b c d message

0 one 1 2 3.0 4 NaN

1 two 5 6 NaN 8 world

2 three 9 10 11.0 12 foo

Using DataFrame’s to\_csv method, we can write the data out to a comma-separated file:

In [50]: data.to\_csv("examples/out.csv")

In [51]: !cat examples/out.csv

,something,a,b,c,d,message

0,one,1,2,3.0,4,

1,two,5,6,,8,world

2,three,9,10,11.0,12,foo

Other delimiters can be used, of course (writing to sys.stdout so it prints the text result to the console rather than a file):

In [52]: import sys

In [53]: data.to\_csv(sys.stdout, sep="|")

|something|a|b|c|d|message

0|one|1|2|3.0|4|

1|two|5|6||8|world

2|three|9|10|11.0|12|foo

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

In [54]: data.to\_csv(sys.stdout, na\_rep="NULL")

,something,a,b,c,d,message

0,one,1,2,3.0,4,NULL

1,two,5,6,NULL,8,world

2,three,9,10,11.0,12,foo

By defaultboth the row and column labels are written. Both of these can be disabled:

In [55]: data.to\_csv(sys.stdout, index=False, header=False)

one,1,2,3.0,4,

two,5,6,,8,world

three,9,10,11.0,12,foo

You can also write only a subset of the columns, and in an order of your choosing:

In [56]: data.to\_csv(sys.stdout, index=False, columns=["a", "b", "c"])

a,b,c

1,2,3.0

5,6,

9,10,11.0

**Working with Other Delimited Formats**

In [57]: !cat examples/ex7.csv

"a","b","c"

"1","2","3"

"1","2","3"

For any file with a single-character delimiter, you can use Python’s built-in csv module. To use it, pass any open file or file-like object to csv.reader:

In [58]: import csv

In [59]: f = open("examples/ex7.csv")

In [60]: reader = csv.reader(f)

Iterating through the reader like a file yields lists of values with any quote characters removed:

In [61]: for line in reader:

....: print(line)

['a', 'b', 'c']

['1', '2', '3']

['1', '2', '3']

In [62]: f.close()

**Binary Data Formats**

One simple way to store (or serialize) data in binary format is using Python’s built-in pickle module.

pandas objects all have a to\_pickle method that writes the data to disk in pickle format:

In [95]: frame = pd.read\_csv("examples/ex1.csv")

In [96]: frame

Out[96]:

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

In [97]: frame.to\_pickle("examples/frame\_pickle")

Pickle files are in general readable only in Python. You can read any “pickled” object stored in a file by using the built-in pickle directly, or even more conveniently using pandas.read\_pickle:

In [98]: pd.read\_pickle("examples/frame\_pickle")

Out[98]:

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

Pandas has built-in support for several other open source binary data formats, such as HDF5, ORC, and Apache Parquet.

**Reading Microsoft Excel Files**

Pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the pandas.ExcelFile class or pandas.read\_excel function.

Internally,these tools use the add-on packages xlrd and openpyxl to read old-style XLS and newer XLSX files, respectively.

These must be installed separately from pandas using

pip or conda:

pip install openpyxl xlrd

In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")

This object can show you the list of available sheet names in the file:

In [102]: xlsx.sheet\_names

Out[102]: ['Sheet1']

Data stored in a sheet can then be read into DataFrame with parse:

In [103]: xlsx.parse(sheet\_name="Sheet1")

Out[103]:

Unnamed: 0 a b c d message

0 0 1 2 3 4 hello

1 1 5 6 7 8 world

2 2 9 10 11 12 foo

In [105]: frame = pd.read\_excel("examples/ex1.xlsx", sheet\_name="Sheet1")

In [106]: frame

Out[106]:

Unnamed: 0 a b c d message

0 0 1 2 3 4 hello

1 1 5 6 7 8 world

2 2 9 10 11 12 foo

To write pandas data to Excel format, you must first create an ExcelWriter, then write data to it using the pandas object’s to\_excel method:

In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")

In [108]: frame.to\_excel(writer, "Sheet1")

In [109]: writer.save()

You can also pass a file path to to\_excel and avoid the ExcelWriter:

In [110]: frame.to\_excel("examples/ex2.xlsx")

**UNIT-4.**

**Data Cleaning and Preparation**

**Handling Missing Data**

In [14]: float\_data = pd.Series([1.2, -3.5, np.nan, 0])

In [15]: float\_data

Out[15]:

0 1.2

1 -3.5

2 NaN

3 0.0

dtype: float64

The isna method gives us a Boolean Series with True where values are null:

In [16]: float\_data.isna()

Out[16]:

0 False

1 False

2 True

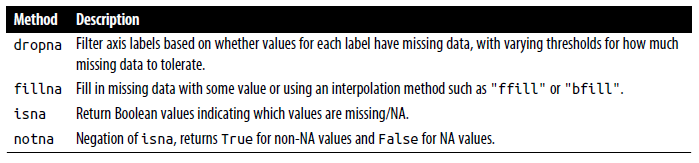
3 False

dtype: bool

It is important to do analysis on the missing data itself to identify data

collection problems or potential biases in the data caused by missing data.

List of some functions related to missing data handling.



**Filtering Out Missing Data**

In [23]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])

In [24]: data.dropna()

Out[24]:

0 1.0

2 3.5

4 7.0

dtype: float64

This is the same thing as doing:

In [25]: data[data.notna()]

Out[25]:

0 1.0

2 3.5

4 7.0

dtype: float64

With DataFrame objects, there are different ways to remove missing data.

In [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],

[np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])

In [27]: data

Out[27]:

0 1 2

0 1.0 6.5 3.0

1 1.0 NaN NaN

2 NaN NaN NaN

3 NaN 6.5 3.0

In [28]: data.dropna()

Out[28]:

0 1 2

0 1.0 6.5 3.0

Passing how="all" will drop only rows that are all NA:

In [29]: data.dropna(how="all")

Out[29]:

0 1 2

0 1.0 6.5 3.0

1 1.0 NaN NaN

3 NaN 6.5 3.0

To drop columns in the same way, pass axis="columns":

In [30]: data[4] = np.nan

In [31]: data

Out[31]:

0 1 2 4

0 1.0 6.5 3.0 NaN

1 1.0 NaN NaN NaN

2 NaN NaN NaN NaN

3 NaN 6.5 3.0 NaN

In [32]: data.dropna(axis="columns", how="all")

Out[32]:

0 1 2

0 1.0 6.5 3.0

1 1.0 NaN NaN

2 NaN NaN NaN

3 NaN 6.5 3.0

Suppose you want to keep only rows containing at most a certain number of missing observations. You can indicate this with the thresh argument:

In [33]: df = pd.DataFrame(np.random.standard\_normal((7, 3)))

In [34]: df.iloc[:4, 1] = np.nan

In [35]: df.iloc[:2, 2] = np.nan

In [36]: df

Out[36]:

0 1 2

0 -0.204708 NaN NaN

1 -0.555730 NaN NaN

2 0.092908 NaN 0.769023

3 1.246435 NaN -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

In [37]: df.dropna()

Out[37]:

0 1 2

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

In [38]: df.dropna(thresh=2)

Out[38]:

0 1 2

2 0.092908 NaN 0.769023

3 1.246435 NaN -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

**Filling In Missing Data**

In [39]: df.fillna(0)

Out[39]:

0 1 2

0 -0.204708 0.000000 0.000000

1 -0.555730 0.000000 0.000000

2 0.092908 0.000000 0.769023

3 1.246435 0.000000 -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

Calling fillna with a dictionary, you can use a different fill value for each column:

In [40]: df.fillna({1: 0.5, 2: 0})

Out[40]:

0 1 2

0 -0.204708 0.500000 0.000000

1 -0.555730 0.500000 0.000000

2 0.092908 0.500000 0.769023

3 1.246435 0.500000 -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

In [41]: df = pd.DataFrame(np.random.standard\_normal((6, 3)))

In [42]: df.iloc[2:, 1] = np.nan

In [43]: df.iloc[4:, 2] = np.nan

In [44]: df

Out[44]:

0 1 2

0 0.476985 3.248944 -1.021228

1 -0.577087 0.124121 0.302614

2 0.523772 NaN 1.343810

3 -0.713544 NaN -2.370232

4 -1.860761 NaN NaN

5 -1.265934 NaN NaN

In [45]: df.fillna(method="ffill")

Out[45]:

0 1 2

0 0.476985 3.248944 -1.021228

1 -0.577087 0.124121 0.302614

2 0.523772 0.124121 1.343810

3 -0.713544 0.124121 -2.370232

4 -1.860761 0.124121 -2.370232

5 -1.265934 0.124121 -2.370232

In [46]: df.fillna(method="ffill", limit=2)

Out[46]:

0 1 2

0 0.476985 3.248944 -1.021228

1 -0.577087 0.124121 0.302614

2 0.523772 0.124121 1.343810

3 -0.713544 0.124121 -2.370232

4 -1.860761 NaN -2.370232

5 -1.265934 NaN -2.370232

With fillna you can do lots of other things such as simple data imputation using the

median or mean statistics:

In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])

In [48]: data.fillna(data.mean())

Out[48]:

0 1.000000

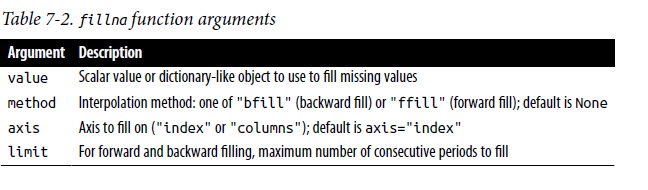
1 3.833333

2 3.500000

3 3.833333

4 7.000000

dtype: float64

  
**Data Transformation**

Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

In [49]: data = pd.DataFrame({"k1": ["one", "two"] \* 3 + ["two"],

....: "k2": [1, 1, 2, 3, 3, 4, 4]})

In [50]: data

Out[50]:

k1 k2

0 one 1

1 two 1

2 one 2

3 two 3

4 one 3

5 two 4

6 two 4

The DataFrame method duplicated returns a Boolean Series indicating whether

each row is a duplicate (its column values are exactly equal to those in an earlier row)

or not:

In [51]: data.duplicated()

Out[51]:

0 False

1 False

2 False

3 False

4 False

5 False

6 True

dtype: bool

In [52]: data.drop\_duplicates()

Out[52]:

k1 k2

0 one 1

1 two 1

2 one 2

3 two 3

4 one 3

5 two 4

**Transforming Data Using a Function or Mapping**

In [57]: data = pd.DataFrame({"food": ["bacon", "pulled pork", "bacon",

....: "pastrami", "corned beef", "bacon",

....: "pastrami", "honey ham", "nova lox"],

....: "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [58]: data

Out[58]:

food ounces

0 bacon 4.0

1 pulled pork 3.0

2 bacon 12.0

3 pastrami 6.0

4 corned beef 7.5

5 bacon 8.0

6 pastrami 3.0

7 honey ham 5.0

8 nova lox 6.0

Suppose you wanted to add a column indicating the type of animal that each food came from. Let’s write down a mapping of each distinct meat type to the kind of animal:

meat\_to\_animal = {

"bacon": "pig",

"pulled pork": "pig",

"pastrami": "cow",

"corned beef": "cow",

"honey ham": "pig",

"nova lox": "salmon"

}

In [60]: data["animal"] = data["food"].map(meat\_to\_animal)

In [61]: data

Out[61]:

food ounces animal

0 bacon 4.0 pig

1 pulled pork 3.0 pig

2 bacon 12.0 pig

3 pastrami 6.0 cow

4 corned beef 7.5 cow

5 bacon 8.0 pig

6 pastrami 3.0 cow

7 honey ham 5.0 pig

8 nova lox 6.0 salmon

We could also have passed a function that does all the work:

In [62]: def get\_animal(x):

....: return meat\_to\_animal[x]

In [63]: data["food"].map(get\_animal)

Out[63]:

0 pig

1 pig

2 pig

3 cow

4 cow

5 pig

6 cow

7 pig

8 salmon

Name: food, dtype: object

**Replacing Values**

In [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [65]: data

Out[65]:

0 1.0

1 -999.0

2 2.0

3 -999.0

4 -1000.0

5 3.0

dtype: float64

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series:

In [66]: data.replace(-999, np.nan)

Out[66]:

0 1.0

1 NaN

2 2.0

3 NaN

4 -1000.0

5 3.0

dtype: float64

If you want to replace multiple values at once, you instead pass a list and then the

substitute value:

In [67]: data.replace([-999, -1000], np.nan)

Out[67]:

0 1.0

1 NaN

2 2.0

3 NaN

4 NaN

5 3.0

dtype: float64

To use a different replacement for each value, pass a list of substitutes:

In [68]: data.replace([-999, -1000], [np.nan, 0])

Out[68]:

0 1.0

1 NaN

2 2.0

3 NaN

4 0.0

5 3.0

dtype: float64

The argument passed can also be a dictionary:

In [69]: data.replace({-999: np.nan, -1000: 0})

Out[69]:

0 1.0

1 NaN

2 2.0

3 NaN

4 0.0

5 3.0

dtype: float64