SMM692
Introduction to Programming in Python

Contents

Preface **9**

1 Organization of the Module **11**

2 Getting Started with Python **13**
- 2.1 Installing Python . . . 13
- 2.2 How Python Runs Programs . . . 13
- 2.3 How We Run Python Programs . . . 13
- 2.4 Managing Python Environments . . . 13

3 Python Language Fundamentals **15**
- 3.1 Variables . . . 15
- 3.2 Objects . . . 15
- 3.3 References . . . 15

4 Python Objects **17**
- 4.1 Number Type Fundamentals . . . 18
- 4.2 String Type Fundamentals . . . 25
- 4.3 List and Dictionaries . . . 33
- 4.4 Dictionaries . . . 37
- 4.5 Tuples . . . 40
- 4.6 Sets . . . 42
- 4.7 Files . . . 44
- 4.8 Python Statements and Syntax . . . 46
- 4.9 Control Flow (or If-Then Statements) . . . 49
- 4.10 While and For Loops . . . 51
- 4.11 Iterations and Comprehensions . . . 55

5 Technical & Scientific Computation with NumPy and SciPy **59**
- 5.1 Installing NumPy and SciPy . . . 59
- 5.2 NumPy `ndarray` . . . 59
- 5.3 Array Creation Routines . . . 65
- 5.4 Array Manipulation Routines . . . 80
- 5.5 Universal Functions in NumPy . . . 82
- 5.6 Mathematical Functions . . . 82
- 5.7 Statistics . . . 84
- 5.8 Linear Algebra . . . 90
- 5.9 Pseudorandom Number Generation . . . 98
- 5.10 File Input and Output with `ndarrays` . . . 98

6 Data Management with Pandas **99**

7 Coda **101**

Appendices

Appendix A Cheat Sheets **105**
A.1 Escapes . . . 105
A.2 String Methods . . . 106
A.3 NumPy Array Manipulation Routines . . . 108

Appendix B Collaborative and Versioning Tools **115**

List of Figures

5.1 A contour plot showing the associations among X, Y, and Z 74
5.2 Visual representation of NumPy `.sin` and `.cos` functions as per Snippet 5.12. 84

List of Tables

4.1 Built-In Objects in Python . . . 18
4.2 Number Type Objects in Python . . . 19
4.3 A Sample of Functions Provided by the `math` Module . . . 20
4.4 Operator Precedence Hierarchy (Ascending Order) . . . 22
4.5 Number Formatting Options in Python . . . 24
4.6 Sample of String Literals and Operators . . . 29
4.7 Popular list methods . . . 35
4.8 Popular dictionary methods . . . 39
4.9 Popular file methods . . . 47
4.10 Python Statements . . . 48

5.1 Common Use Attributes of NumPy `array` . . . 62
5.2 NumPy Data Types . . . 64
5.3 Routines for Creating Arrays from Shape or Value . . . 67
5.4 Routines for Creating Arrays from Existing Data . . . 70
5.5 Routines for Creating Record Arrays . . . 72
5.6 Routines for Numerical Ranges . . . 75
5.7 Routines for Building Matrices . . . 77
5.8 Routines for the Matrix Class . . . 79
5.9 NumPy Stastistical Routines: Order Statistics . . . 85
5.10 numPy Statistical Routines: Average and Variances . . . 86
5.11 NumPy Stastistical Routines: Correlating . . . 87
5.12 NumPy Statistical Routines: Histograms . . . 88
5.13 NumPy Linear Algebra Routines: Matrix and Vector Products . . . 91
5.14 NumPy Linear Algebra Routines: Decompositions . . . 92
5.15 NumPy Linear Algebra Routines: Matrix Eigenvalues . . . 93
5.16 NumPy Linear Algebra Routines: Norms and Other Number . . . 94
5.17 NumPy Linear Algebra Routines: Solving Equations and Inverting Matrices . . . 95

A.1 Helpful Escapes . . . 105
A.2 Comprehensive List of String Methods . . . 106
A.3 NumPy Universal Functions: Mathematical Operations . . . 108
A.4 NumPy Universal Functions: Trigonometric Operations . . . 109
A.5 NumPy Universal Functions: Floating Operations . . . 110
A.6 NumPy Universal Functions: Comparison Operations . . . 111
A.7 NumPy Array Manipulation Routines . . . 112

Preface

...

Chapter 1

Organization of the Module

...

Chapter 2

Getting Started with Python

2.1 Installing Python

...

2.2 How Python Runs Programs

...

2.3 How We Run Python Programs

...

2.4 Managing Python Environments

...

Chapter 3

Python Language Fundamentals

3.1 Variables

...

3.2 Objects

...

3.3 References

...

Chapter 4

Python Objects

In this chapter, we pursue the following learning objectives:

- ....

What is a Python object?

In essence, Python objects are pieces of data. Mark Lutz, the author of the popular book Learning Python[1], points out

> *in Python we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations.*

Built-in and ad-hoc objects

In Python, there are two families of objects: built-in objects provided by the Python language itself and ad-hoc objects — called classes — we can create to accomplish specific goals.

Why do built-in Python objects matter?

Typically, we do not need to create ad-hoc objects. Python provides us with diverse built-in objects that make our job easier:

- built-in objects make coding efficient and easy. For example, using the string object, we can represent and manipulate a piece of text — e.g., a newspaper article — without loading any module
- built-in objects are flexible. For example, we can deploy built-in objects to create a class
- built-in objects have been created and refined over time by a large community of expert developers. Hence, they are often more efficient than ad-hoc objects (unless the creator of the ad-hoc object really knows her business!)

The core built-in Python objects

Table 4.1 illustrates the types of built-in Python objects. For example, Numbers and strings objects are used to represent numeric and textual data respectively. Lists and dictionaries are — likely as not — the two most popular data structures in Python. Lists are ordered collections of other objects such (any type!!). Dictionaries are pairs of keys (e.g., a product identifier) and objects (e.g., the price of the product). No worries: we will go through each built-in type in the following sections of this document. Caveat: in the interest of logical coherence, the various built-in types will not be presented in the order adopted Table 4.1.

TABLE 4.1
Built-In Objects in Python

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation types	Compiled code, stack tracebacks

4.1 Number Type Fundamentals

Types of 'number' objects

Snippet 4.1, "Doing stuff with numbers," highlights the two most popular 'number' instances in Python: integers and floating-point numbers. Integers are whole numbers such as 0, 4, or -12. Floating-point numbers are the representation of real numbers such as 0.5, 3.1415, or -1.6e-19. However, floating points in Python do not have — in general — the same value as the real number they represent.[2] It is worth noticing that any single number with a period '.' is considered a floating point in Python. Also, Snippet 4.1 shows that the multiplication of an integer by a floating point yields a floating point. That happens because Python first converts operands up to the type of the most complicated operand.

Snippet 4.1 — doing 'stuff' with numbers

```
# integer addition
>>> 1 + 1
2

# floating-point multiplication
>>> 10 * 0.5
5.0

# 3 to the power 100
>>> 3 ** 100
515377520732011331036461129765621272702107522001
```

Besides integers and floating points

Besides integers and floating points numbers, Python includes fixed-precision, rational numbers, Booleans, and sets instances — see Table 4.2.

TABLE 4.2
Number Type Objects in Python

Literal	Interpretation
1234, -24, 0, 99999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.X
0177, 0o177, 0x9ff, 0b101010	Octal, octal, hex, and binary literals in 2.X
3+4j, 3.0+4.0j, 3J	Complex number literals
set('spam'), {1, 2, 3, 4}	Sets: 2.X and 3.X construction forms
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

Basic arithmetic operations in Python

Numbers in Python support the usual mathematical operations:

- '+' addition
- '-' → subtraction
- '*' → multiplication
- '/' → floating point division
- '//' → integer division
- '%' → modulus (remainder)
- '**' → exponentiation

To use these operations, it is sufficient to launch a Python or IPython session without any modules loaded (see Snippet 4.1).

Advanced mathematical operations

Besides the mathematical operations shown above, there are many modules shipped with Python that carry out advanced/specific numerical analysis. For example, the `math` module provides access to the mathematical functions defined by the C standard.[3] Table 4.3 reports a sample of these functions. To use them `math`, we have to import the module as shown in Snippet 4.2. Another popular module shipped with Python is `random`, implementing pseudo-random number generators for various distributions (see the lower section of Example 2).

TABLE 4.3
A Sample of Functions Provided by the `math` Module

Function name	Expression
`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	$ln x$
`math.log(x, b)`	$log_b(x)$
`math.log10(x)`	$log_{10}(x)$
`math.sin(x)`	$sin(x)$
`math.cos(x)`	$cos(x)$
`math.tan(x)`	$tan(x)$
`math.asin(x)`	$arcsin(x)$
`math.acos(x)`	$arccos(x)$
`math.atan(x)`	$arctan(x)$
`math.sinh(x)`	$sinh(x)$
`math.cosh(x)`	$cosh(x)$
`math.tanh(x)`	$tanh(x)$
`math.asinh(x)`	$arsinh(x)$
`math.acosh(x)`	$arcosh(x)$
`math.atanh(x)`	$artanh(x)$
`math.hypot(x, y)`	The Euclidean norm, $\sqrt{x^2+y^2}$
`math.factorial(x)`	$x!$
`math.erf(x)`	The error function at x
`math.gamma(x)`	The gamma function at x, $\omega(x)$
`math.degrees(x)`	Converts x from radians to degrees
`math.radians(x)`	Converts x from degrees to radians

Snippet 4.2 — advanced mathematical operations with the modules shipped with Python

```
1 # import the math module
2 >>> import math
3
4 # base-y log of x
5 >>> math.log(12, 8)
6 1.1949875002403856
7
8 # base-10 log of x
9 >>> math.log10(12)
10 1.0791812460476249
11
12 # import the random module
13 >>> import random
14
15 # a draw from a normal distribution with mean = 0 and standard deviation = 1
16 >>> random.normalvariate(0, 1)
17 -0.136017752991189
18
19 # trigonometric functions
20 >>> math.cos(0)
21 1.0
22
23 >>> math.sin(0)
24 0.0
25
26 >>> math.tan(0)
27 0.0
28
29 # an expression containing a factorial product
30 >>> math.factorial(4) - 4 * 3 * 2 * 1
31 0
```

Operator precedence

As shown in Snippet 4.2, line 30, Python expressions can string together multiple operators. So, how does Python know which operation to perform first? The answer to this question lies in operator precedence. When you write an expression with more than one operator, Python groups its parts according to what are called precedence rules,[4] and this grouping determines the order in which the expression's parts are computed. Table 4.4 reports the precedence hierarchy concerning the most common operators. Note that operators lower in the table have higher precedence. Parentheses can be used to create subexpressions that override operator precedence rules.

TABLE 4.4
Operator Precedence Hierarchy
(Ascending Order)

Operator	Description
`x + y`	Addition, concatenation
`x - y`	Subtraction, set difference
`x * y`	Multiplication, repetition
`x % y`	Remainder, format;
`x / y`, `x // y`	Division: true and floor
`-x`, `+x`	Negation, identity
`~x`	Bitwise NOT (inversion)
`x ** y`	Power (exponentiation)

Technical and scientific computation with Python

Python is at the center of a rich ecosystem of modules for technical and scientific computation. In the following chapter, the attention will revolve around two of the most prominent modules: NumPy and SciPy. In a nutshell, NumPy offers the infrastructure for the efficient manipulation of (potentially massive) data structures, while SciPy implements many algorithms across the fields of statistics, linear algebra, optimization, calculus, signal processing, image processing, and others. Another core module in the technical and scientific domain is SimPy, a library for symbolic mathematics. Note that none of these three modules are shipped with Python and should be installed with the package manager of your choice (e.g., `conda`).

Variables and Basic Expressions

Variables are simply names—created by you or Python—that are used to keep track of information in your program. In Python:

- Variables are created when they are first assigned values
- Variables are replaced with their values when used in expressions
- Variables must be assigned before they can be used in expressions
- Variables refer to objects and are never declared ahead of time

As Snippet 4.3 shows, the assignment of `x = 2` causes the variable `x` to come into existence 'automatically.' From that point, we can use the variables in the context of expressions such as the ones displayed in lines 8, 12, 16, and 20, or to create new variables like in line 24.

Snippet 4.3 — expressions involving arithmetic operations

```
1
2  # let us assign the variables 'x' and 'y' to two number objects
3  >>> x = 2
4
5  >>> y = 4.0
6
7  # subtracting an integer from variable 'x'
8  >>> x - 1
9  1
10
11 # dividing the variable 'y' by an integer
12 >>> y / 73
13 0.0547945205479452
14
15 # integer-dividing the variable 'y' by an integer
16 >>> y // 73
17 0.0
18
19 # getting a linear combination of 'x' and 'y'
20 >>> 3 * x - 5 * y
21 -14.0
22
23 # assigning the variable 'z' to the linear combination of 'x' and 'y'
24 >>> z = 3 * x - 5 * y
```

Displaying number objects

Snippet 4.3 includes some expressions whose result is not passed to a new variable (e.g., lines 8, 12, 16, 20). In those cases, the IPython session displays the outcome of the expression 'as is' (e.g., 0.0547945205479452). However, a number with more than three or four decimals may not suit the table or report we have to prepare. Python has powerful string formatting capabilities to display number objects in a readable and nice manner. Table 4.5 illustrates various number formatting options with concrete cases. Format strings contain 'replacement fields' surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. Snippet 4.4 presents a fully-fledged number formatting case. First, we assign the variable `a` to a floating-point number (line 2). Then, we pass the formatting option {`:.2f`} over the variable `a` using the Python built-in function `format`.

Table 4.5
Number Formatting Options in Python

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Snippet 4.4 — number formatting in Python

```
# assign the variable 'a' to a floating-point number
>>> a = 0.67544908755

# displaying 'a' with the first two decimals only
>>> "{:.2f}".format(a)
"0.68"

# displaying 'a' with the first three decimals only
>>> "{:.3f}".format(a)
"0.675"
```

How do I compare number objects?

Comparisons are used frequently to create control flows, a topic we will discuss later in this chapter. Normal comparisons in Python regard two number objects and return a Boolean result. Chained comparisons concern three or more objects, and, like normal comparisons, yield a Boolean result. Snippet 4.5 provides a sample of normal comparisons (between lines 1 and 15) and chained comparisons (between lines 21 and 30). As evident in the example, comparisons can regard both numbers and variables assigned to numbers. Chained comparisons can take the form of a range test (see line 21), a joined, 'AND' test of the truth of multiple expressions (see line 25) or a disjoined, 'OR' test of the truth of multiple expressions (see line 29).

Snippet 4.5 — comparing numeric objects

```
# less than
>>> 3 < 2
False

# greater than or equal
>>> 1 <= 2
True

# equal
>>> 2 == 2
True

# not equal
>>> 4 != 4
False

# range test
>>> x = 3
>>> y = 5
>>> z = 4
>>> x < y < z
False

# joined test
>>> x < y and y > z
True

# disjoined test
>>> x < y or y < z
True
```

4.2 String Type Fundamentals

What is a string?

A Python string is a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are *immutable sequences* of one-character strings; other, more general sequence types include lists and tuples, covered later.

How do we use strings?

Strings are used to record words, contents of text files loaded into memory, Internet addresses, Python source code, and so on. Strings can also be used to hold the raw bytes used for media files and network transfers, and both the encoded and decoded forms of non-ASCII Unicode text used in internationalized programs.

Is `abc` a Python string?

Nope. Python strings are enclosed in single quotes ('...') *or* double quotes ("...") with the same result. Hence, "`abc`" can be Python string, while `abc` cannot. `abc` can be a variable name, though.

String indexing and slicing

The fact that strings are immutable sequences affects how we manipulate textual data in Python. In Snippet 4.6, we fetch the individual elements of `S`, a variable assigned to "`Python 3.X`." As per the built-in function `len`, `S` contains six unitary strings. That means that each element in `S` is associated with a position in the numerical progression $\{0, 1, 2, 3, 4, 5\}$. Now, you may be surprised to see the first element of the list is 0 instead of 1. The reason is that Python is a zero-based indexed programming language: the first element of a series has index 0, while the last element has index `len(obj) - 1`. Fetching the individual elements of a string, such as `S`, requires passing the desired index between brackets, as shown in line 9 (where we get the first unitary string, namely, "`P`".), line 13 (where we get the last unitary string, namely, "`X`"), and line 21 (where we get the unitary string with index 3, i.e., the fourth unitary string appearing in `S`, "`h`"). Note that line 17 is an alternative indexing strategy to the one presented in line 13: it is possible to retrieve the last unitary string by counting 'backward'; that is, getting the first element starting from the right-hand side of the string, which equates to index `-1`. In lines 26 and 30, we exploit the indices of `S` to retrieve multiple unitary strings in a row. What we pass among brackets is not a single index. Instead, we specify a range of indices `i:j`. It is wort noticing that, in Python, the element associated with the lower bound index `i` is returned, whereas the element associated with the upper bound index `j` is not. In line 26, we fetch the unitary strings between index `2` — equating to third unitary string of `S` — and index `5` excluded — namely, the fifth unitary string of `S`. In line 30, we adopt the 'backward' approach to retrieve the unitary string with index `-3` — the third string counting from the right-hand side of `S` — as well as any other unitary strings following index `-3`. To do that, we leave the upper bound index blank.

Snippet 4.6 — Python strings as sequences

```
# let us assign the string "Python 3.X" to the variable S
>>> S = "Python 3.X"

# check the length of S
>>> len(S)
6

# access the first unitary string in the sequence behind S
>>> S[0]
"P"

# access the last unitary string in the sequence behind S
>>> S[len(S)-1]
"X"

# or, equivalently
>>> S[-1]
"X"

# access the i-th, e.g., 3rd, unitary string in the sequence behind S
>>> S[3]
"h"

# access the unitary strings between the i-th and j-th positions in the
# sequence behind S
>>> S[2:5]
"tho"

# access the unitary strings following the i-th position in the sequence
# behind S
>>> S[-3:]
"3.X"
```

Common string literals and operators

Snippet 4.6 deals with string indexing and slicing, two of the many operations we can carry out on strings. Table 4.6 reports a sample of common string literals and operators. The first two lines of Table 4.6 remind us that single and double quotes are equivalent when it comes to assigning a variable to a string object. However, we must refrain from mixing and matching single and double quotes. In other words, a string object requires the leading and trailing quotes are of the same type (i.e., double-double or single-single). In the interest of consistency, it is a good idea to make a policy choice, such as "in my Python code, I use double quotes only", and to stick with that throughout the various lines of the script. I prefer using double quotes because the single quote symbol is relatively popular in natural language (consider for example the Saxon genitive). As shown in the third line of Table 4.6 the single quote is treated as a unitary string insofar as double quotes are used to delimit the string object. Should the string object be delimited by single quotes, we should tell Python not to treat the single quote symbol after `m` as a Python special character but as a unitary string. To do that, we use the escape symbol \ as shown in the fourth line of Table 4.6. Table A.1 provides several escaping examples.

Table 4.6
Sample of String Literals and Operators

Literal/operation	Interpretation
`S = ""`	Empty string
`S = ‘’`	Single quotes, same as double quotes
`S = "spam’s"`	Single quote as a string
`S = ‘spam\’s’`	Escape symbol
`length(S)`	Length
`S[i]`	Index
`S[i:j]`	Slice
`S1 + S2`	Concatenate
`S * 3`	Repeat S n times (e.g., three times)
`"text".join(strlist)`	Join multiple strings on a character (e.g., “text”)
`"{}".format()`	String formatting expression
`S.strip()`	Remove white spaces
`S.replace("pa", "xx")`	Replacement
`S.split(",")`	Split on a character (e.g., “,”)
`S.lower()`	Case conversion — to lower case
`S.upper()`	Case conversion — to upper case
`S.find("text")`	Search substring (e.g., ”text”)
`S.isdigit()`	Test if the string is a digit
`S.endswith("spam")`	End test
`S.startswith("spam")`	Start test
`S = """...multiline..."""`	Triple-quoted block strings

String manipulation tasks

Snippet 4.7 presents a sample of miscellaneous string manipulation tasks. In lines 6 and 9, we check the length of the variables `S1` and `S2`. In line 13, we display five repetitions of `S1`. In line 17, we use the algebraic operator “+” to concatenate `S1` and `S2`. In line 21, we expand on the previous input by separating `S1` and `S2` by a white-space. In line 25, we carry out the same task as line 17 — however, we rely on the built-in `join` function to join `S1` and `S2` with whitespace. The argument taken by `join` is a Python `list`, the subject of paragraph 5.3. In line 29, `S1` and `S2` are joined with a custom string object, namely, `" Vs.  "`. Finally, in line 33, we use the built-in `format` function (see also Snippet 4.4) to display a string object including `S1` and `S2`. For a comprehensive list of string methods, see Table A.2.

Snippet 4.7 — miscellaneous string manipulation tasks

```
# let us assign S1 and S2 to two strings
>>> S1 = "Python 3.X"
>>> S2 = "Julia"

# check the length of S1 and S2
>>> len(S1)
10

>>> len(S2)
5

# display the S1 repeated five times
>>> S1 * 5
"Python 3.XPython 3.XPython 3.XPython 3.XPython 3.X"

# display the concatenation of S1 and S2
>>> S1 + S2
"Python 3.XJulia"

# display the concatenation of S1, whitespace, and S2
>>> S1 + " " + S2
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with a whitespace
>>> " ".join([S1, S2])
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with an arbitrary string object
>>> " Vs. ".join([S1, S2])
"Python 3.X Vs. Julia"

# string formatting
>>> "Both {} and {} have outstanding ML modules".format(S1, S2)
"Both Python 3.X and Julia have outstanding ML modules"
```

String editing

Snippet 4.8 illustrates some string editing tasks. In line 5, we use lstrip — a variation of the built-in function strip — that returns a copy of the string with leading characters removed. In line 9, we use the built-in replace to return a copy of the string with all occurrences of substring `old` (first argument taken by the function) replaced by `new` (second argument taken by the function). Finally, in line 17, we use the built-in function `lower` to return a copy of the string with all the cased characters converted to lowercase.

Snippet 4.8 — miscellaneous string editing tasks

```
1  # let us assign S to a string object
2  >>> S = "Both Python 3.X and Julia have outstanding ML modules"
3  
4  # strip target leading characters
5  >>> S.lstrip("Both ")
6  "Python 3.X and Julia have outstanding ML modules"
7  
8  # replace target characters
9  >>> S.replace("Python 3.X", "R")
10 "Both R and Julia have outstanding ML modules"
11 
12 # split string on target characters
13 >>> S.split(" and ")
14 ["Both Python 3.X", "Julia have outstanding ML modules"]
15 
16 # make the string lower case
17 >>> S.lower()
18 "both python 3.x and julia have outstanding ml modules"
```

Testing and searching strings

Snippet 4.9 presents a series of string test and search tasks. The built-in function `find` (see lines 5 and 9) returns the lowest index in the string where substring sub is found within the slice `S[start:end]` or -1 if substring is not found. The built-in function `isdigit` return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Finally, the built-in function endswith returns `True` if the string ends with the specified suffix, otherwise returns `False`.

Snippet 4.9 — miscellaneous string test and search tasks

```
1  # let us assign S to a string object
2  >>> S = "The first version of Python was released in 1991"
3
4  # search for "Python" in S
5  >>> S.find("Python")
6  21
7
8  # search for "Julia" in S
9  >>> S.find("Julia")
10 -1
11
12 # slice the string the get Python's release year information
13 >>> SS = S[-4:]
14
15 # display SS
16 >>> SS
17 "1991"
18
19 # test if all characters in SS are digits
20 >>> SS.isdigit()
21 True
22
23 # test if all characters in SS are digits
24 >>> SS.isdigit()
25 True
26
27 # test if all characters in SS are digits
28 >>> SS.isdigit()
29 True
30
31 # test if S ends with "1991" / SS
32 >>> S.endswith(SS)
33 True
```

Multiline string printing

In the previous examples, we came across the built-in function `print`. Such a function can be used to print both number- and string-type objects. Sometimes, what we want to print fits into a single line. In other circumstances, we are interested in visualizing rich data, which can span multiple lines. Snippet 4.9 how to print objects across multiple lines with the triple-quoted block string (see line). As evident from the Python code included in lines 8-13, any line between triple-quotes is considered part of the same string object.

Snippet 4.9 — multiline string printing

```
1  # single-line print
2  >>> print("Hello world!")
3  Hello world!
4  
5  # multi-line print
6  >>> print(
7  ... """
8  ... ========================================================
9  ... COL A        | COL B       | ...         | COL K
10 ... --------------------------------------------------------
11 ... Sheldon      | Cooper      | ...         | bazinga.com
12 ... --------------------------------------------------------
13 ... NOTES: this table has fake data
14 ... """
15 ... )
16 
17 ========================================================
18 COL A        | COL B       | ...         | COL K
19 --------------------------------------------------------
20 Sheldon      | Cooper      | ...         | bazinga.com
21 --------------------------------------------------------
22 NOTES: this table has fake data
```

4.3 List and Dictionaries

What is a list?

A Python `list` is an *ordered*, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, `[]`.

Why do we use lists?

Lists are just places to collect other objects so you can treat them as groups.

What type of objects can we include in a list?

Lists can contain any sort of object: numbers, strings, and even other lists. See Snippet 4.10.

Snippet 4.10 — sample lists with different items

```
# an empty list
>>> L = []

# a list with an integer, a float, and a string
>>> L = [2, -3.56, "XyZ"]

# a list with an integer and a list
>>> L = [4, ["abc", 8.98]]
```

List indexing

We can retrieve one or more list component objects via indexing. That is possible because list items are ordered by their position (similarly to strings). Since Python is a zero-based indexed programming language, to fetch the first item of a list we have to call the index `0` (see Snippet 4.11, line 5). A list nested in another list can be fetched by using multiple indices (line 13). The first index refers to the outer list, while any subsequent index refers to an inner list. In our case, we have two indices, one for each list; that is, `L` and its sub-list `["abc", 8.98]`.

Snippet 4.11 — list indexing and slicing

```
1  # the list
2  >>> L = [4, ["abc", 8.98]]
3
4  # get the first item of L
5  >>> L[0]
6  4
7
8   # get the second element of L
9  >>> L[1]
10 ["abc", 8.98]
11
12 # get the first item of L's second item
13 >>> L[1][0]
14 "abc"
```

List mutability

Lists are mutable objects, which may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more. Snippet 4.12 illustrates some snippets to change a list's items. In the first part of the example, we change the items using indexing (line 5) and slicing (line 10). In the second half, we use Python's `del` statement to delete the items using indexing (line 15) and slicing (line 20).

Snippet 4.12 — changing and deleting list items in place

```
# the list
>>> L = ["Leonard", "Penny", "Sheldon"]

# change the second item of L via indexing
>>> L[1] = "Raj"
>>> print(L)
["Leonard", "Raj", "Sheldon"]

# change multiple items of L via slicing
>>> L[0:2] = ["Amy", "Howard"]
>>> print(L)
["Amy", "Howard", "Sheldon"]

# delete the first item of L via indexing and using the 'del' statement
>>> del L[0]
>>> print(L)
["Howard", "Sheldon"]

# delete multiple items of L via slicing and using the 'del' statement
>>> del L[0:2]
>>> print(L)
[]
```

List manipulation with built-in methods

Python offers many methods to manipulate and test list objects. Table 4.7 reports some of the most popular methods along with synopses. The first three methods — `.append()`, `.insert()`, and `.extend()` — expand an existing list. The fourth method, `.index` test for the presence of an item in an existing list. Note this method raises a `ValueError` if there is no such item.The remaining methods produce in-place changes in an existing list's items.

TABLE 4.7
Popular list methods

Method	Synopsis
`L.append(X)`	Append an item to an existing list
`L.insert(i, X)`	Append an item to an exisintg list in position i
`L.extend([X0, X1, X2])`	Extend an existing list with the items from another list
`L.index(X)`	Get the index of the first instance of the argument in an existing list
`L.count(X)`	Get the cardinality of an item in an existing list
`L.sort()`	Sort the items in an existing list
`L.reverse()`	Reverse the order of the items in an existing list
`L.copy()`	Get a copy of an existing list
`L.pop(i)`	Remove the item at the given position in the list, and return it
`L.remove(X)`	Remove the first instance of an item in an existing list
`L.clear()`	Remove all items in an existing list

Expanding an existing list

Both the `.append()` and `.extend()` methods can be used to expand an existing list as per Snippet 4.13. However, they accomplish different goals and should not be confused: `.append()` adds a new item (of any type) to the end of the list (see line 6); `.extend()` extend the list by appending all the items from another iterable (e.g., another list, see line 11).

Snippet 4.13 — methods for expanding an existing list

```
1  # create two lists
2  >>> L1 = ["Leonard", "Penny", "Sheldon"]
3  >>> L2 = ["Howard", "Raj", "Amy", "Bernadette"]
4
5  # expand an existing list with .append()
6  >>> L2.append("Priya")
7  >>> print(L2)
8  ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
9
10 # concatenate L1 and L2 with .extend()
11 >>> L1.extend(L2)
12 >>> print(L1)
13 ["Leonard", "Penny", "Sheldon", "Howard", "Raj", "Amy", "Bernadette", "Priya"]
```

In-place change of an existing list's items

One of the most common list manipulation task consists of changing the order of an item's list. As shown in Snippet 4.14, it is possible to use the `.reverse()` method to reverse the elements of the list in place (see line 5), while sorting an item's list can be carried out with the `.sort()` method.

Snippet 4.14 — methods for changing list items in place

```
1  # create a list
2  >>> L = ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
3
4  # reverse the list's item positions
5  >>> L.reverse()
6  >>> print(L)
7  ["Priya", "Bernadette", "Amy", "Raj", "Howard"]
8
9  # sort the list's items
10 >>> L.sort()
11 >>> print(L)
12 ["Amy", "Bernadette", "Howard", "Priya", "Raj"]
```

4.4 Dictionaries

What is a dictionary?

Along with lists, dictionaries are one of the most flexible built-in data types in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by *positional offset*.

Why do we use dictionaries?

Dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

What type of objects can we include in a dictionary?

Like lists, dictionaries can contain objects of any type, and they support nesting to any depth(they can contain lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

How do we create a dictionary?

Snippet 4.15 shows two different ways to create a dictionary. A dictionary can be created by including key-value pairs among braces (see line 2). In the example, there are three keys, associated with Marvel characters, and as many values, which can be thought as the characters' position in an ideal power rank. A colon separates a key and its associated value. The second way to create a dictionary is based on Python's builtin `dict`, mapping key onto values, and `zip`, which iterates over two elements in parallel. Specifically, `zip` creates the one-to-one correspondence between keys (characters) and values (characaters' power) that is passed as the argument of `dict`. We will analyze the topic of iterations extensively in sections 4.10 and 4.11.

Snippet 4.15 — initializing a new dictionary object

```
1 # method 1
2 >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4 # method 2
5 >>> CHARACTERS = ["Captain Marvel", "Living Tribunal", "One-Above-All"]
6 >>> RANK = [3, 2, 1]
7 >>> D = dict(zip(CHARACTERS, RANK))
8 >>> print(D)
9 {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
```

Accessing a dictionary's values

Dictionaries' items cannot be accessed via positional offsets — like lists. Instead, we fetch the individual items by using the dictionary keys as shown in Snippet 4.16 (see line 5). The reference key is passed among brackets. When the dictionary at hand contains nested dictionaries (see line 9), it is possible to concatenate multiple queries, namely, sequences of keys between brackets (see line 21).

Snippet 4.16 — fetching dictionary items

```
1  # the dictionary
2  >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4  # let's fetch Captain Marvel's position in the Marvel characters' power rank
5  >>> D["Captain Marvel"]
6  3
7
8  # a dictionary of dictionaries
9  >>> D = {
10         "Dr. Strange": {
11                 "first_appearance": 1963,
12                 "created_by": "Lee & Ditko"
13                 },
14         "Iron Man": {
15                 "first_appearance": 1963,
16                 "created_by": "Lee, Lieber, Heck & Kirby"
17                 },
18     }
19
20 # let us fetch the creator of Dr. Strange
21 >>> D["Dr. Strange"]["created_by"]
22 "Lee & Ditko"
```

Are dictionaries mutable?

Dictionsries, like lists, are mutable. Thus, we can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The del statement works here, too; it deletes the entry associated with the key specified as an index (see Snippet 4.17).

Snippet 4.17 — dictionary mutability examples

```
# the dictionary
>>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}

# let us change the power rank for Captain Marvel
>>> D["Captain Marvel"] = 12
>>> print(D)
{"Captain Marvel": 12, "Living Tribunal": 2, "One-Above-All": 1}

# let us eliminate the character Living Tribunal
>>> del D["Living Tribunal"]
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1}

# let us add a further character
>>> D["Wanda Maximoff"] = 4
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1, "Wanda Maximoff": 4}
```

Dictionary manipulation with built-in methods

Like for lists, Python offers many `methods to manipulate dictionary objects`. Table 4.8 reports some of the most common methods along with synopses. The first three methods, `.keys()` `.values()` `.items()`, get the constitutive elements of dictionaries: keys, values, and key-value pairs respectively. The fourth method, `.get(key, default?)` gets the value for a specific key. The fifth method, `.update()`, updates the value for a specific key. Like `.update()`, `.popitem()`, `.pop()`, and `d.clear()` alter the information of a dictionary in place. The first removes the value of a certain key; the second removes the item (a key-value pair) for a certain key; the latter delete all dictionary items. Finally, `.copy()` creates a shallow copy of an existing dictionary.

TABLE 4.8
Popular dictionary methods

Method	Synopsis
`D.keys()`	Get all dictionary keys
`D.values()`	Get all dictionary values
`D.items()`	Get all dictionary key-value pairs as tuples
`D.get(key, default?)`	Query a dictionary element by key
`D.update(D2)`	Update a dictionary key's value
`D.popitem()`	Remove the value corresponding to a certain key
`D.pop(key, default?)`	Remove the item at the given position in the list,
`D.clear()`	Delete all dictionary items
`D.copy()`	Copy the target dictionary

How do we access the information in a dictionary?

Snippet 4.18 shows how to use builtin methods to carry out three fundamental tasks: accessing dictionary keys (see line 5), values (see line 9), and items (i.e., key-value pairs, see line 13). It is worth noticing that the three methods illustrated in the example yield specific dictionary objects such as `dict_keys`, `dict_values`, and `dict_items`. Translating one of these dictionary objects into a list — if needed — is straightforward (see line 17).

Snippet 4.18 — accessing the information included in a dictionary

```
1 # the dictionary
2 >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4 # get the keys
5 >>> D.keys()
6 dict_keys(["Captain Marvel", "Living Tribunal", "One-Above-All"])
7
8 # get the values
9 >>> D.values()
10 dict_values([3, 2, 1])
11
12 # get the items
13 >>> D.items()
14 dict_items([("Captain Marvel", 3), ("Living Tribunal", 2), ("One-Above-All", 1)])
15
16 # get the keys as a list
17 >>> list(D.keys())
18 ["Captain Marvel", "Living Tribunal", "One-Above-All"]
```

4.5 Tuples

What is a tuple?

Tuples are sequences of immutable Python objects. They are similar to lists, but they are immutable. Tuples are created by enclosing a comma-separated list of values in parentheses.

Tuples are immutable!

Tuples are immutable, which means that once they are created, they cannot be changed!!

Why do we use tuples?

Tuples are useful for storing data that is not to be changed, such as the coordinates of a point in a two-dimensional space. In general, we use tuples any time information integrity is a concern — in other words when we want to make sure the information included in an object will not change because of another reference somewhere in our program.

How do we create a tuple?

Python objects, separated by a comma, must be included between parentheses (see Snippet 4.19, line 2).

How do we access the information in a tuple?

By positional offsets, like lists (see Snippet 4.19, lines 5 and 9).

Snippet 4.19 — creating and accessing a tuple

```
1  # the tuple
2  >>> T = ("Captain Marvel", 3)
3
4  # access a tuple element
5  >>> T[0]
6  "Captain Marvel"
7
8  # access a tuple element
9  >>> T[1]
10 3
```

Can we convert a tuple into a list?

Yes, we can. To do that, we pass the tuple as the argument of `list` (see Snippet 4.20).

Snippet 4.20 — tuple conversion

```
>>> T = ("Captain Marvel", 3)

# from a tuple to a list
>>> L = list(T)
>>> print(L)
["Captain Marvel", 3]

# amend L's items
>>> L[1] = 4

# get back to a tuple
>>> T = tuple(L)
>>> print(T)
("Captain Marvel", 4)
```

Tuples with the `collections` module

`collections` is a module that is shipped with Python and provides data containers that are alternative to Python's general purpose built-in containers, i.e., `dict`, `list`, `set`, and `tuple`. One of these containers can be created with the function `namedtuple` (see Snippet 4.21), which allows annotating the tuple items with names. In line 2, we import the function `namedtuple` from the `collections` module. In line 5, we create an ad hoc class that best represents the structure of our sample data, concerning Marvel characters' names and the year in which they first appeared in the comic series. The first argument taken by the function is customary and regards the name of the class we are about to create. The second argument is a list with the names of the attributes included in our data structure. In line 8, we use the newly created class `Rec` to create a tuple, which is eventually printed as per line 11.

Snippet 4.21 — creating an annotated tuple with the collection module

```
1  # import the namedtuple function from the module collection
2  >>> from collections import namedtuple
3
4  # create an ad hoc class object 'Rec' that fits our data structure
5  >>> Rec = namedtuple("Rec", ["character", "first_appearance"])
6
7  # use the generated class "Rec"
8  >>> IRONMAN = Rec("Iron Man", 1963)
9
10 # A named-tuple record
11 >>> IRONMAN
12 Rec(character="Iron Man", first_appearance=1963)
```

4.6 Sets

What is a set?

A `set` is an *unordered* collection of *unique* and *immutable* objects.

What does it mean that sets are unordered collections?

By design, `set` is a data structure with *undefined element ordering* (see Snippet 4.22 — the outcome included in line 6 does not follow any particular order).

What does it mean that sets have unique items?

By definition, an item appears only once in a set, no matter how many times it is added (see Snippet 4.22, line 2 Vs. line 7).

Snippet 4.22 — creating a set

```
# create a list
>>> L = ["a", "a", "b", "c", "c"]

# get a set from L
>>> S = set(L)
>>> print(S)
{"b", "a", "c"}
```

Why do we use sets?

Sets made this way support common mathematical set operations (see Snippet 4.23). Hence, they have a variety of applications, especially in numeric and database-focused work.

Snippet 4.23 — set operations

```
# create two sets
>>> X = set(["a", "b", "c"])
>>> Y = set(["c", "d", "e"])

# set difference
>>> X - X
set()
>>> X - Y
{"a", "b"}

# union
>>> X | Y
{"a", "b", "c", "d", "e"}

# intersection
>>> X & Y
{"c"}

# superset
>>> X > Y
False

# subset
>>> X < Y
False
```

4.7 Files

How do the files in our OS relate with Python?

Our Python program may involve input and/or output operations. In other words, we may want to read data from a file stored in our machine and/or write the outcome of our analysis to a file. The built-in function `open` creates a Python file object, which serves as a link to a file residing on your machine. As Lutz notes:

> *"Compared to the types you've seen so far, file objects are somewhat unusual. They are considered a core type because they are created by a built-in function, but they're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks" (page 282)*

How do we open a file?

We open a pipe to a file using the built-in function `open`. The output of the function is a `file` object.

How do we source the data stored in a file?

We open a pipe to a file using the built-in function `open`. The output of the function is a `file` object. Snippet 4.24 illustrates how to use `open` for data sourcing. In the first part of the snippet, we create a `file` object to read the data included in the existing file `my_file.txt`.[5] At least, we have to pass one argument to `open`: the path pointing to the file. A second optional argument is `mode`, which specifies the mode in which the file is opened to source. It defaults to `r`, which means open for reading in text mode. Other common values are `w` for writing,[6] `x` for exclusive creation, and `a` for appending. [7] To read a file's contents, we use the `.read()` method (see line 9), returning a string object (see line 10).

Snippet 4.24 — data input with open

```
1  # create a pipe to a file
2  >>> file = open(file="my_file.txt", mode="r")
3
4  # calling "file" yields the attributes of the file object
5  >>> file
6  <_io.TextIOWrapper name="my_file.txt" mode="r" encoding="UTF-8">
7
8  # let us source the data
9  >>> data = file.read()
10 >>> print(data)
11 Hi there
12
13 # close the pipe
14 >>> file.close()
```

How do we write the data in the current Python session to a file?

Snippet 4.25 illustrates how to use `open` for data writing. In the first part of the snippet, we create three strings — i.e., the information we are manipulating in the active Python session (see lines 2, 4, and 6). Then, we create a file object in 'writing' mode (see the value passed to `mode`, line 9). Finally, we manipulate the three strings (as a sample task, in line 12, we concatenate `FIRSTLAW`, `SECONDLAW`, `THIRDLAW`) and write the result to a file (line 16).

Snippet 4.25 — data output with open

```
1  # the strings (data) to save permanently to a file
2  >>> FIRSTLAW = "A robot may not injure a human being or, through inaction, "\
3                 "allow a human being to come to harm."
4  >>> SECONDLAW = "A robot must obey the orders given it by human beings except "\
5                 "where such orders would conflict with the First Law."
6  >>> THIRDLAW = "A robot must protect its own existence as long as such "\
7                 "protection does not conflict with the First or Second Law."
8
9  # create a pipe to a file
10 >>> file = open(file="my_file.txt", mode="w")
11
12 # concatenate the strings
13 >>> TO_WRITE = "\n".join([FIRSTLAW, SECONDLAW, THIRDLAW])
14
15 # write the concatenated strings
16 >>> file.write(TO_WRITE)
17
18 # close the pipe
19 >>> file.close()
```

How about reading a single line from a file?

Hold on: what is a line? A string whose last character is `\n`. We can read a single line from a file using the `.readline()` method (see Snippet 4.26). Such a method starts by reading the first line included in the file (see line 11); then, it reads any subsequent lines included in the file (see line 15); when it reaches the end of the file (EOF), it returns the empty string `""` (see line 19).

Snippet 4.26 — reading one line at a time with `.readline()`

```
# the strings (data) to save permanently to a file
>>> DATA = "The first line\nThe second line"

# create a pipe to a file and write DATA
>>> file = open(file="my_file.txt", mode="w")
>>> file.write(DATA)
>>> file.close()

# read one line from the file
>>> file = open(file="my_file.txt", mode="r")
>>> file.readline()
"The first line\n"

# calling file.readline() again reds the subsequent line
>>> file.readline()
"The second line"

# ... and so on until the end of the file is reached
>>> file.readline()
""
```

How about reading multiple lines at a time?

The `.readlines()` method reads the lines from a file and returns them as a list (see Snippet 4.27).

Snippet 4.27 — reading multiple lines at a time with `.readlines())`

```
# the strings (data) to save permanently to a file
>>> DATA = "A\nB\nC\nD"

# create a pipe to a file and write DATA
>>> file = open(file="my_file.txt", mode="w")
>>> file.write(DATA)
>>> file.close()

# read multiple lines
>>> file = open(file="my_file.txt", mode="r")
>>> file.readlines()
['A\n', 'B\n', 'C\n', 'D']

# equivalently to the previous line, we can use 'list'
['A\n', 'B\n', 'C\n', 'D']
```

What are the most common file methods?

Table 4.9 illustrates some key file methods' names and their corresponding synopsis.

4.8 Python Statements and Syntax

Table 4.9
Popular file methods

Method	Description
`file.close()`	Closes the file
`file.detach()`	Returns the separated raw stream from the buffer
`file.fileno()`	Returns a number that represents the stream as per the OS' perspective
`file.flush()`	Flushes the internal buffer
`file.isatty()`	Returns whether the file stream is interactive or not
`file.read()`	Returns the file content
`file.readable()`	Returns whether the file stream can be read or not
`file.readline()`	Returns one line from the file
`file.readlines()`	Returns a list of lines from the file
`file.seek()`	Change the file position
`file.seekable()`	Returns whether the file allows us to change the file position
`file.tell()`	Returns the current file position
`file.truncate()`	Resizes the file to a specified size
`file.writable()`	Returns whether the file can be written to or not
`file.write()`	Writes the specified string to the file
`file.writelines()`	Writes a list of strings to the file

Notes: `file` is a fictionary object used to illustrate the usage of the file methods.

What is a Python statement?

In his popular book 'Learning Python,' Lutz provides a concise and effective description of what a Python statement is:

> *In simple terms, statements are the things you write to tell Python what your programs should do. If, as suggested [omitted], programs "do things with stuff," then statements are the way you specify what sort of things a program does. Less informally, Python is a procedural, statement-based language; by combining statements, you specify a procedure that Python performs to satisfy a program's goals.*

What are Python's statements?

Table 4.10 illustrates common Python statements, their role, and application examples. Some of these statements were used in the examples considered so far. Other statements — the majority — will be faced in the next sections of the current chapter and/or in the subsequent chapters.

TABLE 4.10
Python Statements

Statement	Role	Example
`import`	Module access	`import math`
`from`	Attribute access	`from math import sqrt`
`class`	Building ad hoc objects	`class Subclass(Superclass):  def method(self):  pass`
`del`	Deleting references	`del a`
Assignment	Creating references	`a = "before b"`
Calls and other expressions	Running functions	`file.write("Hello")`
`print`	Printing objects	`print("Hello")`
`if/elif/else`	Selecting actions	`if "abc" in text:  print(text)`
`for/else`	Iteration	`for x in mylist:  print(x)`
`while/else`	General loops	`while X > Y: print("Hello")`
`pass`	Empty placeholder	`while True:  pass`
`break`	Loop exit	`while True:  if exit test():  break`
`continue`	Loop continue	`while True:  if skiptest():  continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):  for i in n:  yield i*2`

4.9 Control Flow (or If-Then Statements)

What is control flow in Python?

Many Python statements we write are compound statements: there is one statement nested inside another. The outer statement is called the 'if' statement and the inner statement is called the 'then' statement. The 'if' statement is used to determine whether to execute the 'then' statement. Specifically, the 'then' statement is executed insofar as the 'if' statement evaluates to 'True.' Snippet 4.28 illustrates a control flow case, a simple rule-based product recommender that suggests products based on users' purchasing patterns. If product `x` belongs to a user's set of past purchases, then a certain item is recommended; otherwise, no recommendation is offered (see lines 7 and 8, containing the `else` statement).

Snippet 4.28 — an example of control flow in Python

```
1 # a set with a customer's past purchases
2 >>> S = set(["a", "x", "u"])
3 
4 # a rule-based product recommender
5 >>> if "x" in S:
6 ...     print("Customers who bought x also bought Air Jordan 7 Retro Miro")
7 ... else:
8 ...     pass
9 Customers who bought x also bought Air Jordan 7 Retro Miro
```

End of line → end of the statement

Any Python statements are contained in the same line — the end of the line equates to the statement end. In the interest of redundancy, Python statements do not traverse multiple lines. In Snippet 4.28, the if statement is in line 5; the then statement is in line 6.

The ':' character is required for nested statements!

The colon character is required to separate the if statement from the 'then' statement (see Snippet 4.28 line 5).

Indentation has substantive meaning in Python!

'Then' statements are indented (with a tab or four consecutive spaces). Do not creatively use indents to embellish your code — that is not consistent with Python's rules and design principles (see Snippet 4.28 line 6).

End of indentation → end of nested statements

‘Then’ statements are indented (with a tab or four consecutive spaces). In Snippet 4.28, the indentation in line 6 makes lines 5 and 6 to be evaluated together.

Multiple ‘then’ statements

Exmple 4.29 shows how to use **elif** to concatenate multiple ‘if-then’ statements in the same control flow. Like in Snippet 4.28, the ‘else’ statement defines the residual behavior of the control flow; that is, what Python does when both the ‘if’ and ‘elif’ statements evaluate to ‘False.’

Snippet 4.29 — an example of control flow with multiple ‘if-then’ statements in Python

```
# a list with a customer's past purchases
>>> S = set(["a", "w", "u"])

# a rule-based product recommender
>>> if "x" in S:
...     print("Customers who bought x also bought Air Jordan 7 Retro Miro")
... elif "w" in S:
        print("How about Converse Chuck Taylor All Star?")
... else:
        print("Falling short of suggestions --- I'm a dull recommender!")
How about Converse Chuck Taylor All Star?
```

Nested if-then statements

In Python, it is possible to nest an if-then statement into another. In Snippet 4.30, the ‘if’ statements in lines 6 and 8 are nested inside the ‘if’ statement in line 5. It is worth noticing that lines 7 and 9 — regarding ‘then’ statements — are indented twice because they terminate distinct if-then statements nested in the broader if-then statement commencing on line 5.

Snippet 4.30 — an example of nested control flow in Python

```
1  # a list with a customer's past purchases
2  >>> S = set(["a", "x", "b"])
3
4  # a rule-based product recommender
5  >>> if "x" in S:
6  ...     if "a" in S:
7  ...         print("Customers who bought x & a also bought Air Force")
8  ...     elif "u" in S:
9  ...         print("Customers who bought x & u also bought Air Max 95")
10 ... else:
11 ...    print("Falling short of suggestions --- I'm a dull recommender!")
12 Customers who bought x & a also bought Air Force
```

4.10 While and For Loops

Looping!

Oftentimes, we write Python statements that repeat a same task — i,e., they loop a certain number of times or over multiple items.

How do we write loops in Python?

Using `for` and `while` statements.

What is the difference between `for` and `while` statements?

The `while` statement provides a way to code general loops. The `for` statement is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

Snippet 4.31 — `while` loop examples

```
# loop until reaching a numeric threshold
>>> i = 0
>>> while i <= 3:
...     print(i)
...     i = i + 1
0
1
2
3

# loop until an empty string is returned
>>> x = "Indiana Jones"
>>> while x != "":
...     print(x)
...     x = x[1:]
...
Indiana Jones
ndiana Jones
diana Jones
iana Jones
ana Jones
na Jones
a Jones
 Jones
Jones
ones
nes
es
s
```

`while` loops in action

`while` statements run a code block insforar as a test evaluates to True. In the upper section of Exmple 4.31, we assing a `i` to a number. Then we create a for loop with the following elements: the first one is a statementing testing wether `i` is smaller or equal 3 (see line 3); the second element is the loop body (indented), which is repeated as long as the test evaluates to `True`. It is worth noticing that every iteration of the loop body produces a unitary increase in `i` — therefore, the program leaves the loop after four iterations. In the lower section of Snippet 4.31, we assigng the variable `x` to a string (line 13), which we eventually print (line 14) and slice (line 15) until we get an empty string (line 13x).

`for` loops in action

`for` steps through a sequence of items and carries out a task. In the upper section of Snippet 4.32, we print the result of a mathematical operation that is deployed over the items of a `list` (an example of Python iterable object). The code included in line 2 assigngs temporarily the variable `item` to an element of iterable. Then, the code block (indented) is executed over the temporary object. In the lower section of Snippet 4.32, the execution of the loop operates a mathematical expression over the items of a first list and appends the outcome to a second list (line 10).

Snippet 4.32 — a `for` loop example

```
1  # print the result of a mathematical operation carried out over a list of items
2  >>> for item in [0, -99, 13, 6.54]:
3  ...     print(item ** 0.5)
4  
5  # run a mathematical operation on a list of items and append the outcome
6  # to a second list
7  >>> input = [2, 8, 1]
8  >>> output = []
9  >>> for item in input:
10 ...     output.append(item + 1)
11 >>> print(output)
12 [3, 9, 2]
```

`for` loops with dictionaries

Like lists, dictionaries are iterable objects. In the upper section of Snippet 4.33, we create a dictionary and iterate over its items printing a simple predicate. As we know from section 4.4, we access a dictionary's values by keys. Hence, in line 5, we retrieve the keys of `D`. Then, in line 12, we fetch the value of the temporary object `k`, namely, `D[k]`. Particularly, we print the temporary object `k`, the string object `IS`, and the value associated with `k`; that is, `D[k]`. In Snippet 34, we accomplish the same task of Snippet 33. However, the loop regards a dictionary's items — i.e., key-value pairs — instead of keys (that is self-evident from the comparison of Snippet 33's line 55 and Snippet 34's line 5).

Snippet 4.33 — looping on dictionary keys

```
1  # the dictionary
2  >>> D = {"Thor": "Asgardian", "Vision": "android", "Wanda Maximoff": "human"}
3
4  # get the keys of D
5  >>> keys = D.keys()
6  >>> print(keys)
7  dict_keys(['Thor', 'Vision', 'Wanda Maximoff'])
8
9  # iterate over the keys to fetch the dictionary values and do something
10 # with them
11 >>> for k in keys:
12 ...     print(k + " IS " + D[k])
13 Thor IS Asgardian
14 Vision IS android
15 Wanda Maximoff IS human
```

Snippet 4.34 — looping on dictionary items

```
1  # the dictionary
2  >>> D = {"Thor": "Asgardian", "Vision": "android", "Wanda Maximoff": "human"}
3
4  # get the items of D
5  >>> items = D.items()
6
7  # iterate over key-value pairs and do something with them
8  >>> for k, v in items:
9  ...     print(k + " IS " + v)
10 Thor IS Asgardian
11 Vision IS android
12 Wanda Maximoff IS human
```

Counter `for` loops

The built-in class `range` provides an immutable sequence that is particularly helpful for loops that repeat an action a certain number of times. Snippet 35 shows an example of a for loop with `range`.

Snippet 4.35 — a counter loop example

```
# show the outcome of range
>>> list(range(3))
[0, 1, 2]

# use range in a for loop
>>> for i in range(3):
...     print(i, ":-)")
0 :-)
1 :-)
2 :-)
```

Nested `for` loops

A Python statement that contains multiple for loops is a nested `for` loop. Mainly, a `for` loop allows to carry out a task over the elements of two iterables jointly. The outer loop considers the individual items of the first iterable (see Snippet 4.26, line 6); the inner loop (indented) considers the individual items of the second list (see line 7). Once we have created a pair of temporary objects, we can do something with it (see line 8).

Snippet 4.36 — a nested loop example

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # create all permutations of letters and colors and print them
6  >>> for i in LETTERS:
7  ...     for j in COLORS:
8  ...         print(i, " - ", j)
9  x  <->  blue
10 x  <->  green
11 x  <->  red
12 y  <->  blue
13 y  <->  green
14 y  <->  red
15 z  <->  blue
16 z  <->  green
17 z  <->  red
```

Nested `for` loops Vs. `zip` for loops

Contrarily to the nested `for` loops, which considers all permutations containing multiple iterables' items, the built-in zip steps through several iterables *in parallel*, producing tuples with an item from each one. As shown in Snippet 4.37, there is neither an inner nor an outer for loop in this case — instead, there is a single loop considering two temporary objects, `i` and `j`, that occupy the same position in the offset of the iterables at hand (see line 6; the first item from the first iterable goes with the first item from the second iterable, the second item from the first iterable goes with the second item from the second iterable, and so on and so forth).

Snippet 4.37 — looping over two iterables in parallel with `zip`

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # create one-to-one matches of items and do something with them
6  >>> for i, j in zip(LETTERS, COLORS):
7  ...     print(i, " <-> ", j)
8  x  <->  blue
9  y  <->  green
10 z  <->  red
```

4.11 Iterations and Comprehensions

Over and beyond `while` and `for` loops

As we know from the previous section, `while` and `for` loops can handle most repetitive tasks programs need to perform. However, Python provides further tools to make loops *easier to write/read* and *more efficient*. One of the most prominent tools is `list comprehension`.

Why do we use `list comprehensions?`

To create a list containing the outcome of an action repeated over an iterable's items (see Snippet 4.38, line 14).

How do we create `list comprehensions`?

We include a Python statement containing a `for` clause among brackets (see Snippet 4.38, line 12).

Snippet 4.38 — `for` loop Vs. list comprehension

```
# the for loop way
# --+ create an empty list
L = []
# --+ create a for loop appending the square of some items
>>> for i in range(3):
...     L.append(i ** 2)
# --+ print the list
>>> print(L)
[0, 1, 4]

# the list comprehension way
>>> L = [i ** 2 for i in range(3)]
>>> print(L)
[0, 1, 4]
```

How do we implement a nested `for` loop in list comprehensions?

As Snippet 4.39 shows, a nested for loop becomes a one-liner in a list comprehension. The first `for` clause in line 7 would correspond to the outer `for` loop reported in Snippet 4.36, whereas the second `for` clause in line 7 would correspond to the inner `for` loop reported in Snippet 4.36.

Snippet 4.39 — nested for loop with list comprehensions

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4  
5  # implementing a nested for loop with a list comprehension
6  >>> LETTER2COLOR = ["{} <-> {}".format(i, j) for i in LETTERS for j in COLORS]
7  ['x <-> blue',
8   'x <-> green',
9   'x <-> red',
10  'y <-> blue',
11  'y <-> green',
12  'y <-> red',
13  'z <-> blue',
14  'z <-> green',
15  'z <-> red']
```

Can we use the `zip` generator within a list comprehension?

Yes, we can. To do that, the `for` clause must consider two iterables at once (see Snippet 4.40, line 6).

Snippet 4.40 — looping over two iterables in parallel with `zip` and list comprehension

```
1 # the lists
2 >>> LETTERS = ["x", "y", "z"]
3 >>> COLORS = ["blue", "green", "red"]
4
5 # implementing a nested for loop with a list comprehension
6 >>> LETTER2COLOR = ["{} <-> {}".format(i, j) for i, j in zip(LETTERS, COLORS)]
7 ['1 <-> blue', '2 <-> green', '3 <-> red']
```

Can we use embed control flow in a list comprehension?

Yes, we can. To do that, the `for` clause must be preceded by an `if` statement and, at least, an `else` statement (see for example Snippet 4.40's line 22).

Snippet 4.41 — control flow in list comprehensions

```
1 # import the function log from math
2 from math import log
3
4 # the object to manipulate
5 >>> L1 = [0, 1, 2]
6
7 # the for loop way
8 # --+ the empty list
9 L2 = []
10 # --+ the for loop appending the log of some items
11 >>> for i in L1:
12 ...     if i > 0:
13 ...         L2.append(log(i))
14 ...     else:
15 ...         L2.append(log(i + 0.001))
16 # --+ print the list
17 >>> print(L2)
18 [-6.907755278982137, 0.0, 0.6931471805599453]
19
20 # the list comprehension way
21 # --+ the list comprehension is a one-liner!
22 >>> L2 = [log(i) if i > 0 else log(i + 0.001) for i in L1]
23 # --+ print the list
24 >>> print(L2)
25 [-6.907755278982137, 0.0, 0.6931471805599453]
```

Notes

[1]Lutz, Mark. *Learning Python: Powerful object-oriented programming.* O'Reilly Media, Inc., 2013.

[2]Floating numbers are stored in binaries with an assigned level of precision that is typically equivalent to 15 or 16 decimals.

[3]As per the documentation of the Python programming language, `math` cannot be used with complex numbers.

[4]The official Python documentation has an extensive section on operator precedence rules in the section dedicated to syntax of expressions

[5]For the sake of simplicity, we assume the target file is located in the same directory as the Python script.

[6]By deafult, `w` truncates the file if it already exists

[7]If encoding is not specified the encoding used is platform-dependent. Specifically, `locale.getpreferredencoding(False)`

is called to get the current locale encoding. Character encoding is the process of assigning numbers to graphical characters, especially the written characters of human language, allowing them to be stored, transmitted, and transformed using digital computers.

Chapter 5

Technical & Scientific Computation with NumPy and SciPy

Learning goals:

-
-

5.1 Installing NumPy and SciPy

Do NumPy and SciPy come with the official Python installation file?

No, they do not. You need to install them separately using the package manager `pip`.

I am an Anaconda user: do I need to install NumPy and SciPy separately?

No, you do not. NumPy and SciPy are included in 'base', the default environment of Anaconda. However, if you create a new environment, you need to install NumPy and SciPy — as well as all the other modules you need — with the package manager `conda`.

How do I install NumPy and SciPy?

The easiest way is using the command line. Anaconda users run `$ conda install numpy scipy`, whereas Python official release users run `$ pip install numpy scipy`. Having said that, Anaconda users can also install the modules they need from within Anaconda-Navigator.

5.2 NumPy `ndarray`

What is a NumPy `ndarray`?

Put simply, an `ndarray` is a data container, like dictionaries and lists.

Can `ndarrays` contain objects of different type?

No, they cannot. An `ndarray` must contain homogenous items; that is, items of the same type.

How do we create an `ndarray`?

As shown in Snippet 5.1, we pass an object to `numpy.array`. If the object we pass is a scalar, a 0-dimensional array containing object is returned (line 5). Passing a list to `numpy.array` produces a one-dimensional array (line 9); passing a list of lists produces a two-dimensional array (line 13); finally, passing a list of lists of lists produces a three-dimensional array (line 18).

Snippet 5.1 — creating an `ndarray`

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # a 0-D array
5  >>> np.array(0)
6  array(0)
7
8  # a 1-D array
9  >>> np.array([1, 2, 3, 4])
10 array([1, 2, 3, 4])
11
12 # a 2-D array
13 >>> np.array([[1, 2], [3, 4]])
14 array([[1, 2],
15        [3, 4]])
16
17 # a 3-D array
18 >>> np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
19 array([[[1, 2],
20         [3, 4]],
21
22        [[5, 6],
23         [7, 8]]])
```

What are the distinctive features of `ndarrays`?

`ndarrays` have been designed — and tuned over time — with flexibility and efficiency in mind. For example, `ndarrays` allow to carry out computations on arrays with a syntax similar to scalar values. As shown in Snippet 5.2, we can multiply an array by a scalar (line 10) as well as add two vectors (14). That is not possible if we use pure Python code. Multiplying a list by a scalar N replicates the ordered collection of items N times (see line 23). Adding two lists yields concatenation (see line 30). At the same time, `ndarrays` support the analysis of large volumes of data[8].

Snippet 5.2 — NumPy allows to manipulate vectors with an expressive syntax

```
1 # import numpy with the socially accepted alias 'np'
2 import numpy as np
3
4 # generate some random
5 >>> DATA = np.random.randn(3)
6 >>> print(DATA)
7 [-0.44144029 -0.44451097  0.31997294]
8
9 # can we multiply a list by a scalar with NumPy? Of course!
10 >>> print(DATA * 3)
11 [-4.4144029 , -4.44510974,  3.19972941]
12
13 # can we sum two arrays with NumPy? Of course!
14 >>> print(DATA + DATA)
15 [-0.88288058, -0.88902195,  0.63994588]
16
17 # let us try to replicate the previous tasks in pure Python?
18 # --+ get the DATA as a list
19 >>> DATA = list(DATA)
20 >>> print(DATA)
21 [-0.4414402896845323, -0.4445109735283278, 0.31997294069261617]
22 # --+ is the NumPy syntax of line 10 still valid if we use a list? Nope
23 >>> print(DATA * 3)
24 [
25 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
26 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
27 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617
28 ]
29 # --+ is the NumPy syntax of line 14 still valid if we use a list? Nope
30 >>> print(DATA + DATA)
31 [
32 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
33 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617
34 ]
```

How do I check an `array`'s number of dimensions?

In Snippet 5.3, we saw NumPy infers an `ndarrays`'s number of dimensions from the data. Snippet 5.3 shows how to access `.ndim`, the `ndarrays`'s attribute concerning the number of dimensions. In the example, `DATA` has two dimensions (e.g., coordinates).

How do I check an `array`'s shape?

The lower section of Snippet 5.3 shows how to access `.shape`, the `ndarrays`'s attribute concerning the shape. In the example, each dimension of the `DATA` has size 2.

Snippet 5.3 — an array's number of dimensions and shape

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the data
>>> DATA = np.array([[1, 2], [3, 4]])
>>> print(DATA)
[[1 2]
 [3 4]]

# get the number of dimensions
>>> DATA.ndim
2

# get the shape
>>> DATA.shape
(2, 2)
```

What are the attributes of an `array`?

Table 5.1 illustrates the common use attributes of `ndarrays`.

TABLE 5.1
Common Use Attributes of NumPy `array`

Attribute	Synopsis
`DATA.flags`	Information about the memory layout of the array
`DATA.shape`	Tuple of array dimensions
`DATA.strides`	Tuple of bytes to step in each dimension when traversing an array
`DATA.ndim`	Number of array dimensions
`DATA.data`	Python buffer object pointing to the start of the array's data
`DATA.size`	Number of elements in the array
`DATA.itemsize`	Length of one array element in bytes
`DATA.nbytes`	Total bytes consumed by the elements of the array
`DATA.dtype`	Data-type of the array's elements

Notes: `DATA` is a fictionary object used to illustrate the usage of the `array` attributes.

I know that a NumPy array must contain homogenous data — but which object types are allowed?

Table 5.2 reports the NumPy data types. Python beginners are not supposed to appreciate the distinctive attributes of each type. Instead, they may want to get a clear understanding of the high-level types, namely, floating points, complex, integer, boolean, string, or general Python objects. When it comes to working on sophisticated projects, requiring more control over the storage types, then it is highly suggested to get a thorough knowledge of the types in Table 5.2. It is worth noticing that dtypes are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases, they provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating-point value takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`.

How do I specify the data type of an `array`?

As Snippet 5.4 shows, dtype is an optional argument of `ndarray`. It is also possible to change dtype using the method `.astype()`.

Snippet 5.4 — specifying a nd changing dtype

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# accept the default type
>>> A = np.array([1, 2, 3, 4, 5])

# check the type
>>> A.dtype
dtype('int64')

# specify the type
>>> A = np.array([1, 2, 3, 4, 5], dtype=np.int32)

# check the type
>>> A.dtype
type('int32')

# type change
>>> S = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
>>> S = S.astype(float)
>>> S.dtype
dtype('float64')
```

TABLE 5.2
NumPy Data Types

Type	Type Code	Synopsis
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point float32 f4 or f Standard single-precision floating point; compatible with C float
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python
float object	float128 f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

5.3 Array Creation Routines

Does NumPy offer recipes for creating arrays?

Yes, it does. NumPy has seven families of array-creating routines:

- From shape or value
- From existing data
- Creating record arrays (`np.rec`)
- Numerical ranges
- Building matrices
- The Matrix class

Creating arrays from shape or value

This family of routines creates arrays with a certain number of dimensions, shape, values, and attributes. Snippet 5.5. shows how to create:

- an array with a certain shape and a constant scalar (see lines 5 — `.zeros`, 12 — `.ones`, 19 — `full`), and
- an array containing an identity matrix (see lines 26 — `.eye` — and 33 — `.identity`)

Snippet 5.5 — creating arrays from shape or value

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# create an array with zeros only
>>> np.zeros([4,4])
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])

# create an array with ones only
>>> np.ones((4,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])

# create a full matrix with a given scalar
>>> np.full((4,4), -99)
array([[-99, -99, -99, -99],
       [-99, -99, -99, -99],
       [-99, -99, -99, -99],
       [-99, -99, -99, -99]])

# create an identity array of a given shape with .eye
>>> np.eye(4, 3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])

# create an identity array with .identity
>>> np.identity(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

TABLE 5.3
Routines for Creating Arrays from Shape or Value

Routine	Synopsis
`np.empty(shape[, dtype, order, like])`	Return a new array of given shape and type, without initializing entries
`np.empty_like(prototype[, dtype, order, subok, ...])`	Return a new array with the same shape and type as a given array
`np.eye(N[, M, k, dtype, order, like])`	Return a 2-D array with ones on the diagonal and zeros elsewhere
`np.identity(n[, dtype, like])`	Return the identity array
`np.ones(shape[, dtype, order, like])`	Return a new array of given shape and type, filled with ones
`np.ones_like(a[, dtype, order, subok, ...])`	Return an array of ones with the same shape and type as a given array
`np.zeros(shape[, dtype, order, like])`	Return a new array of given shape and type, filled with zeros
`np.full(shape, fill_value[, dtype, order, like])`	Return a new array of given shape and type, filled with fill value
`full_like(a, fill_value[, dtype, order, ...])`	Return a full array with the same shape and type as a given array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Creating arrays from existing data

In Snippets 5.1 – 5.4, we saw how to use `array` for passing data to a NumPy array. Snippet 5.6 shows further routines to create NumPy arrays from existing data include for example:

- `.fromfunction`, creating an array by executing a function over each coordinate (line 8)
- `.fromfile`, creating an array from data in a text or binary file (line 19)
- `.loadtxt`, loading data from a text file (line 37). The example represents a real-world data set containing both numeric and text information. The first argument we pass `.loadtxt` is a file object. To correctly parse the data, we also pass the following discretionary arguments to `.loadtxt`: i) `comments="#"` indicates that any lines in the file commencing with `#` must be considered a comment, not a piece of data; ii) `delimiter=","` indicates that two items separated by the character `,` belong to different fields (i.e., ‘columns’ to use a spreadsheet-alike vocabulary); iii) texttttquotechar=”’ indicates that strings are enclosed between double quotes[9].

Snippet 5.6 — creating arrays from existing data

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # get data from a function
5  # --+ create a function
6  >>> my_function = lambda x, y: x - 0.5 * y ** 2
7  # --+ create an array from my for given coordinates
8  >>> np.fromfunction(my_function, (3, 3), dtype=float)
9  array([[ 0.  , -0.5, -2. ],
10        [ 1.  ,  0.5, -1. ],
11        [ 2.  ,  1.5,  0. ]])
12
13 # get data from a binary file
14 # --+ create an array from a list of numbers
15 >>> D = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])))
16 # --+ save the raw data to a binary file
17 >>> D.tofile("data.bin")
18 # --+ read the data back
19 >>> np.fromfile("data.bin", dtype=int)
20
21 # get data from a text file
22 # --+ create a string with the data and some qualitative comments on them
23 >>> S = """
24 # Below are some demographic data about Michael J. Jordan (basketball player)
```

```
# from Wikipedia.
#
# Data labels are:
#
# NAME, BORN, NBA CHAMPIONSHIPS, AVERAGE POINT PER GAME
"Jordan, Michael Jeffrey","17-02-1963",6,30.1
"""
# --+ write the data to a file
>>> with open("my_data", "w") as pipe:
...     pipe.write(S)
>>> pipe.close()
# --+ read the data and assign them to a NumPy array
>>> np.loadtxt(
...   open("my_data", "r"),
...       dtype={
...             "names": (
...                       "NAME",
...                       "BORN",
...                       "NBA CHAMPIONSHIPS",
...                       "AVERAGE POINT PER GAME"
...                            ),
...             "formats": ("S30", "S10", "i1", "f2"),
...             },
...       comments="#",
...       delimiter=",",
...       quotechar='"'
...   )
array((b'Jordan, Michael Jeffrey', b'17-02-1963', 6, 30.1),
       dtype=[('NAME', 'S30'),('BORN', 'S10'),
             ('NBA CHAMPIONSHIPS', 'i1'),
             ('AVERAGE POINT PER GAME', '<f2')]
)
```

TABLE 5.4
Routines for Creating Arrays from Existing Data

Routine	Synopsis
`np.array(object[,dtype, copy, subok, ...])`	Create an array
`np.asarray(a[, dtype, order, like])`	Convert the input to an array
`np.asanyarray(a[, dtype, order, like])`	Convert the input to an ndarray, but pass ndarray subclasses through
`np.ascontiguousarray(a[, dtype, like])`	Return a contiguous array (ndim ¿= 1) in memory (C order)
`np.asmatrix(data[, dtype])`	Interpret the input as a matrix
`np.copy(a[, order, subok])`	Return an array copy of the given object
`np.frombuffer(buffer[, dtype, count, offset, like])`	Interpret a buffer as a 1-dimensional array
`np.fromfile(file[, dtype, count, sep, offset, like])`	Construct an array from data in a text or binary file
`np.fromfunction(function, shape, *[, dtype, like])`	Construct an array by executing a function over each coordinate
`np.fromiter(iter, dtype[, count, like])`	Create a new 1-dimensional array from an iterable object
`np.fromstring(string[, dtype, count, like])`	A new 1-D array initialized from text data in a string
`np.loadtxt(fname[, dtype, comments, delimiter, ...])`	Load data from a text file

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Record arrays

NumPy arrays do no contain any information about the attributes of the data. For example, a NumPy array cannot accommodate any meta-data, such as the names of the fields included in the data. Here is where `.rec` kick in (see Table 5.5). For example, .core.records.array allows to flexibly specify a field's type and name (see Snippet 5.7, line 8). Once a 'recarray' is created, it is possible to fetch its data by field name (see Snippet 5.7, line 11).

Snippet 5.7 — creating record arrays

```
1  # import records array with an alias that does not conflict with
2  # `standard' NumPy arrays
3  >>> from numpy.core.records import array as recarray
4
5  # the data
6  >>> LOCS = [("51.5072° N", "0.1276° W"), ("35.6762° N", "139.6503° E")]
7
8  # create a recarray
9  >>> D = recarray(LOCS, formats=["U12", "U12"], names=["Latitude", "Longitude"])
10
11 # fetch the data by field name
12 >>> D.Latitude
13 array(['51.5072° N', '35.6762° N'], dtype='<U12')
```

TABLE 5.5
Routines for Creating Record Arrays

Routine	Synopsis
`np.core.records.array(obj[, dtype, shape, ...])`	Construct a record array from a wide-variety of objects
`np.core.records.fromarrays(arrayList[, dtype, ...])`	Create a record array from a (flat) list of arrays
`np.core.records.fromrecords(recList[, dtype, ...])`	Create a recarray from a list of records in text form
`np.core.records.fromstring(datastring[, dtype, ...])`	Create a record array from binary data
`np.core.records.fromfile(fd[, dtype, shape, ...])`	Create an array from binary file data

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Creating numerical ranges

One may want to create a numerical range for different reasons, including running functional analysis or computer simulation. NumPy has a bunch of array-creating routines for numerical ranges (see Table 5.6), some of which that are quite popular in the fields of technical and scientific computation as well as data science. For example, `np.arange` and `np.linspace` frequently appear in Python programs when it comes to create evenly spaced values in a certain interval and evely spaced samples respectively (see Snippet 5.8, lines XX and XX). Another popular routine is `.meshgrid`, returning coordinate matrices from coordinate vectors.

Snippet 5.8 — creating numerical ranges

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# two ranges of evenly spaced values
# --+ evenly spaced values between 0 and 10
>>> np.arange(0, 10, 1)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# --+ ... equivalent to
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# --+ evenly spaced values between 0 and 10 divided by a 2-unit step
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])

# 50 evenly spaced values between 0 and 1
>>> np.linspace(0, 1, 10)
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])

# get coordinate matrices from coordinate vectors
# --+ the 'x-' and 'y-axis' vectors
>>> X = np.linspace(0, 1, 10)
>>> Y = np.linspace(0, 1, 5)
# --+ get 'x-axis' ('y-axis') coordinates for any value of vector Y (X)
>>> XX, YY = np.meshgrid(X, Y)
>>> XX
array([[0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ]])
```

```
37 >>> YY
38 array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
39        [0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25],
40        [0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
41        [0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75],
42        [1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  ]])
43 # --+ create a matrix from X and Y
44 >>> ZZ = np.sqrt(XX**2 + YY**2)
45 # --+ check the dimensions of the newly created objects
46 >>> print(XX.shape, YY.shape, ZZ.shape)
47 >>> ((101, 101), (101, 101), (101, 101))
48 # --+ make a contour plot showing the associations among X, Y, and Z
49 # (see Figure 5.1)
50 >>> fig = plt.figure()
51 >>> ax = fig.add_subplot(111)
52 >>> ax = plt.contourf(X, Y, ZZ)
53 >>> plt.axis('scaled')
54 >>> plt.colorbar()
55 >>> plt.show()
```

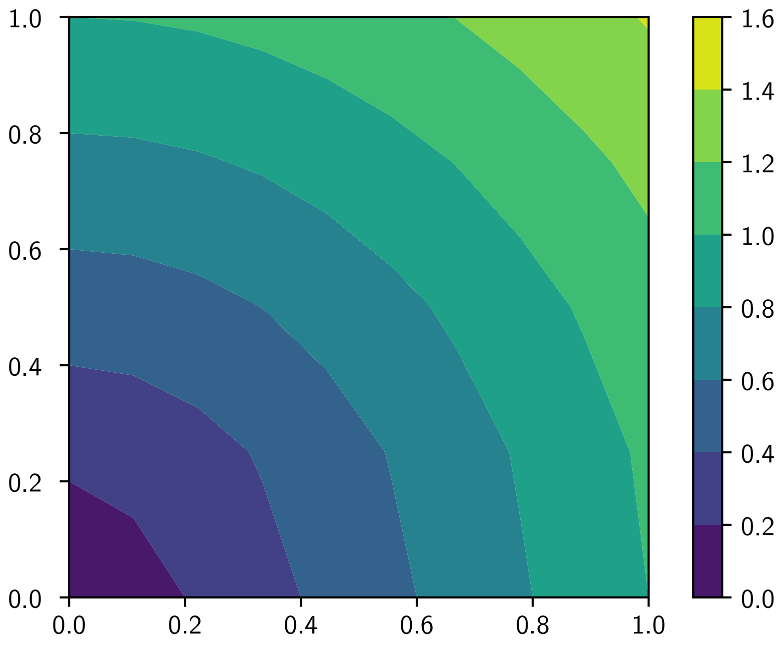


Figure 5.1: A contour plot showing the associations among X, Y, and Z

Notes: the data behind this plot come from Snippet 5.8, lines 22 - 44.

Table 5.6
Routines for Numerical Ranges

Routine	Synopsis
`np.arange([start,] stop[, step,][, dtype, like])`	Return evenly spaced values within a given interval
`np.linspace(start, stop[, num, endpoint, ...])`	Return evenly spaced numbers over a specified interval
`np.logspace(start, stop[, num, endpoint, base, ...])`	Return numbers spaced evenly on a log scale
`np.geomspace(start, stop[, num, endpoint, ...])`	Return numbers spaced evenly on a log scale (a geometric progression)
`np.meshgrid(*xi[, copy, sparse, indexing])`	Return coordinate matrices from coordinate vectors
`np.mgrid`	`nd_grid` instance which returns a dense multi-dimensional "meshgrid"
`np.ogrid`	`nd_grid` instance which returns an open multi-dimensional "meshgrid"

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Building matrices

NumPy has routines to create arrays from an existing matrix as well as build matrices with certain properties (see Table 5.7) As Snippet 5.9 shows, `.diag` creates an array by fetching a matrix's diagonal (see line 10), while `.tri` creates a triangular matrix (see line 17).

Snippet 5.9 — creating arrays from existing data

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # create an array by fetching a matrix diagonal
5  # --+ the matrix
6  >>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
7  >>> M.shape
8  (3, 3)
9  # --+ the new array
10 >>> A = np.diag(M)
11 >>> print(A)
12 [1 5 9]
13 >>> A.shape
14 (3,)
15
16 # create a triangular matrix
17 >>> np.tri(10, 10)
18 array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
19        [1., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
20        [1., 1., 1., 0., 0., 0., 0., 0., 0., 0.],
21        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
22        [1., 1., 1., 1., 1., 0., 0., 0., 0., 0.],
23        [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
24        [1., 1., 1., 1., 1., 1., 1., 0., 0., 0.],
25        [1., 1., 1., 1., 1., 1., 1., 1., 0., 0.],
26        [1., 1., 1., 1., 1., 1., 1., 1., 1., 0.],
27        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

TABLE 5.7
Routines for Building Matrices

Routine	Synopsis
`np.diag(v[, k])`	Extract a diagonal or construct a diagonal array
`np.diagflat(v[, k])`	Create a two-dimensional array with the flattened input as a diagonal
`np.tri(N[, M, k, dtype, like])`	An array with ones at and below the given diagonal and zeros elsewhere
`np.tril(m[, k])`	Lower triangle of an array
`np.triu(m[, k])`	Upper triangle of an array
`np.vander(x[, N, increasing])`	Generate a Vandermonde matrix

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

The Matrix Class

In Snippet 5.9, line 17, I claim to create a matrix. Actually, the outcome displayed in line 18 is consistent with the concept of 'matrix' we have been taught in a typical linear algebra class: what a human being sees is a set of numbers arranged in rows and columns. However, in NumPy terms, the object displayed in line 18 is an array with two dimensions. If we want to create a NumPy Matrix Class object[10], we have to call `np.matrix`, a subclass of `np.ndarray`. Table 5.8 illustrates the two matrix-creating routines of NumPy.

What is the advantage of using a matrix class object?

As we will see later on in this chapter, the advantage of using a matrix class object is that we can use the matrix class object to perform matrix operations with a simple and intuitive syntax. For example, we can use the matrix class object to perform matrix manipulation multiplication, addition, and subtraction. In the lower section of Snippet 5.10, we use `.linalg.inv` to compute the inverse of a matrix.

Snippet 5.10 — creating a NumPy Matrix Class object

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# create a matrix class object
# --+ the arrays to populate a matrix class object
>>> A = np.array([0, 1, 2])
>>> B = np.array([-99, 203, 1009])
>>> C = np.array([-1000, -1001, -1002])
# --+ the matrix class object
>>> M = np.matrix([A, B, C])
matrix([[    0,     1,     2],
        [  -99,   203,  1009],
        [-1000, -1001, -1002]])

# get the inverse of M
>>> np.linalg.inv(M)
matrix([[-1.60040278e+00,  1.98412698e-03, -1.19642857e-03],
        [ 2.19880556e+00, -3.96825397e-03,  3.92857143e-04],
        [-5.99402778e-01,  1.98412698e-03, -1.96428571e-04]])
```

TABLE 5.8
Routines for the Matrix Class

Routine	Synopsis
`np.mat(data[, dtype])`	Interpret the input as a matrix
`bmat(obj[, ldict, gdict])`	Build a matrix object from a string, nested sequence, or array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

5.4 Array Manipulation Routines

I have an array. What can I do with it?

NumPy offers many array manipulation routines (see Table A.7), which can be grouped around the following families:

- basic operations
- changing array shape
- transpose-like operations
- changing number of dimensions
- changing kind of array
- joining arrays
- splitting arrays
- tiling arrays
- adding and removing elements
- rearranging elements

Presenting every NumPy array manipulation routine would require writing a dedicated book (and certainly falls beyond the remit of an introductory module on Python). That said, let me whet the reader's appetite by illustrating a sample of miscellaneous routines (see Snippet 5.11). Here is the sequence of tasks we accomplish: first, we create an array of shape `(6,)` (see line 5); then, we change the shape of the array (line 10) and transpose its rows and columns (line 16); in the lower section of the snippet we join two arrays using several routines: `.concatenate` joins two arrays along the desired axis (the default axis is the first one, meaning the second arrays are concatenated row-wise); `.vstack` requires to pass a tuple of arrays to stack row-wise; `.hstack` requires to pass a tuple of arrays to stack column-wise. It is worth noticing that the rules for vector/matrix manipulation we learned at school apply to the routines that join arrays. In other words, two NumPy arrays can be joined if and only have the same length on the joining axis. For example, it is possible to stack vertically two arrays with shape `(3, 3)` and `(2, 3)` because they have the same number of columns.

Snippet 5.11 — miscellaneous array-manipulating routines

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the array
>>> A = np.arange(6)
>>> A
array([0, 1, 2, 3, 4, 5])

# reshape the array into a 2x3 array
>>> A = A.reshape(2, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5]])

# transpose the array
>>> A.T
array([[0, 3],
       [1, 4],
       [2, 5]])

# get back to a 'flat' array with shape (6,)
>>> A.ravel()
array([0, 1, 2, 3, 4, 5])

# reshape the array into a 3x2 array
>>> A.reshape(3, 2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# join two arrays --- note the dimensions for the concatenation must match
>>> np.concatenate((A.reshape(3, 2), A.T))
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 3],
       [1, 4],
       [2, 5]])

# stack two arrays vertically (ROW-WISE)
>>> np.vstack((A, np.array([6, 7, 8])))

# stack two arrays horizontally (COLUMN-WISE)
>>> np.hstack((A, np.array([6, 7]).reshape(2, 1)))
array([[0, 1, 2, 6],
       [3, 4, 5, 7]])
```

5.5 Universal Functions in NumPy

What is a universal function?

A universal function, or `ufunc`, is a function that performs element-wise operations on `ndarrays`. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results[11].

What is the rationale for using `ufuncts`?

Per the previous point, using a `ufunc` offer substantial performance advantages *vis a' vis* non-vectorized code — i.e., code using built-in Python iterators.

What are the `unfunct` options available in NumPy?

There are circa sixty universal functions implemented in NumPy. For the sake of convenience, the full list of `ufunc` options is reported in Tables A.3 - A.6. The following sections of the current chapter will illustrate how to use some of the popular central `ufuncts` in NumPy.

5.6 Mathematical Functions

What are the mathematical functions available in NumPy?

There are many mathematical functions available in NumPy, some of which are implemented as universal functions (for a complete list of universal functions, see Tables A.3). The available routines can be grouped into:

- trigonometric functions
- hyperbolic functions
- rounding functions
- sums, products, and differences
- exponential and logarithmic functions
- extrema finding

What are the mathematical functions available in NumPy?

Similar to the case of array-manipulating routines, a detailed discussion of every NumPy mathematical function exceeds the scope of these notes. However, every function can be accessed using the same procedure; mainly, we need a set of values to pass to the argument of the function — that is it! Snippet 5.12 shows the procedure to use a NumPy mathematical functions by means of two trignometric functions, namely, `.sin` and `.cos`. In line 9, we create a range of values to assign to the variable `X`; then, in line 11, we create two further arrays assigned to the outcome of `.sin` and `.cos` respectively; finally, we use the Matplotlib module to visualize the two functions (if you do not get the Matplotlib code logic, do not worry at all — you will familiarize with it in the Fall Term module 'SMM635, Data Visualization.')

Snippet 5.12 — NumPy trigonometric functions in action

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # import a data viz module
5  >>> import matplotlib.pyplot as plt
6
7  # trigonometric functions
8  # --+ x-values
9  >>> X = np.arange(0, 2 * np.pi, 0.# --+ plot SI
10 # --+ y-values
11 >>> SI, CS = np.sin(X), np.cos(X)
12
13 # plot the functions
14 # --+ plot SI
15 >>> fig = plt.figure(figsize=(2.5, 2.5))
16 >>> ax = fig.add_subplot(111)
17 >>> ax.axhline(y=0, color="k", linewidth=0.5)
18 >>> ax.set_xticks([0, 0.5 * np.pi, np.pi, 1.5 * np.pi, 2 * np.pi])
19 >>> ax.set_xticklabels(
20 ...     ["0", r"$\frac{1}{2} \pi$", r"$\pi$", r"$\frac{3}{2} \pi$", r"$2 \pi$"]
21 ... )
22 >>> plt.xlabel("$X$")
23 >>> plt.ylabel("$sine(X)$")
24 >>> ax.grid(True)
25 >>> ax.plot(X, SI, color="Blue")
26 >>> plt.title("A")
27 >>> plt.show()
28 # --+ plot CS
29 >>> fig = plt.figure(figsize=(2.5, 2.5))
30 >>> ax = fig.add_subplot(111)
31 >>> ax.axhline(y=0, color="k", linewidth=0.5)
32 >>> ax.set_xticks([0, 0.5 * np.pi, np.pi, 1.5 * np.pi, 2 * np.pi])
33 >>> ax.set_xticklabels(
```

```
...     ["0", r"$\frac{1}{2} \pi$", r"$\pi$", r"$\frac{3}{2} \pi$", r"$2 \pi$"]
... )
>>> plt.xlabel("$X$")
>>> plt.ylabel("$cosine(X)$")
>>> ax.grid(True)
>>> ax.plot(X, CS, color="Blue")
>>> plt.title("B")
>>> plt.show()
```

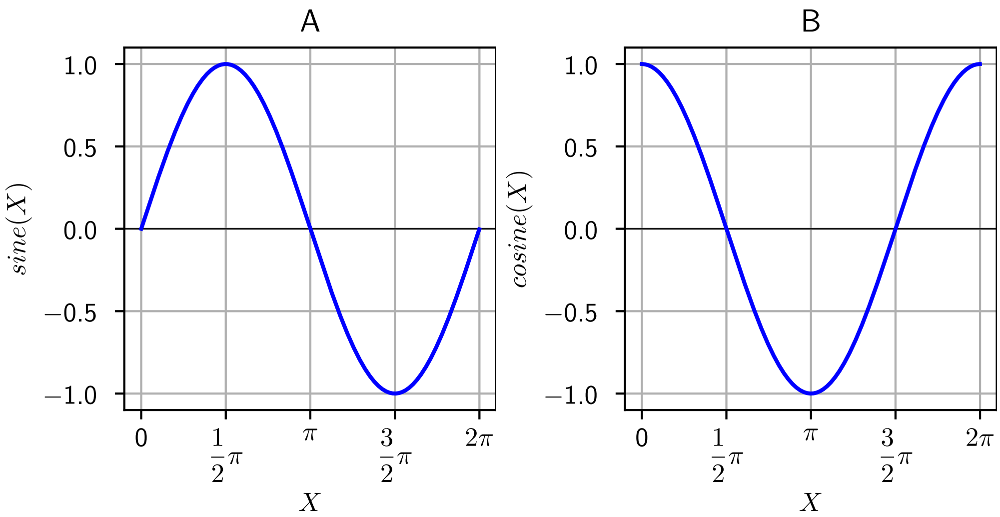


Figure 5.2: Visual representation of NumPy `.sin` and `.cos` functions as per Snippet 5.12.

5.7 Statistics

What are the statistical functions available in NumPy?

NumPy offers essential statistical functions, sufficient to implement an Exploratory Data Analysis/descritive stastistics. Specifically, there are four familieis of statistical routines (see Tables 5.9, 5.10, 5.11, and 5.12):

- order statistics (e.g., quantiles)
- average and variances
- correlations
- histograms

Can I run multivariate statistical analysis with NumPy

Short answer: no. There are dedicated Python modules to run multivariate analyses, though. For example linearmodels and statsmodels are two popular modules to carry out econometric models in Python, while scikit-learn is the acclaimed module for machine learning in Python.

TABLE 5.9
NumPy Stastistical Routines: Order Statistics

Routine	Synopsis
`np.ptp(a[, axis, out, keepdims])`	Range of values (maximum - minimum) along an axis
`np.percentile(a, q[, axis, out, ...])`	Compute the q-th percentile of the data along the specified axis
`np.nanpercentile(a, q[, axis, out, ...])`	Compute the qth percentile of the data along the specified axis, while ignoring nan values
`np.quantile(a, q[, axis, out, overwrite_input, ...])`	Compute the q-th quantile of the data along the specified axis
`np.nanquantile(a, q[, axis, out, ...])`	Compute the qth quantile of the data along the specified axis, while ignoring nan values

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.10
numPy Statistical Routines: Average and Variances

Routine	Synopsis
`np.median(a[, axis, out, overwrite_input, keepdims])`	Compute the median along the specified axis
`np.average(a[, axis, weights, returned, keepdims])`	Compute the weighted average along the specified axis
`np.mean(a[, axis, dtype, out, keepdims, where])`	Compute the arithmetic mean along the specified axis
`np.std(a[, axis, dtype, out, ddof, keepdims, where])`	Compute the standard deviation along the specified axis
`np.var(a[, axis, dtype, out, ddof, keepdims, where])`	Compute the variance along the specified axis
`np.nanmedian(a[, axis, out, overwrite_input, ...])`	Compute the median along the specified axis, while ignoring NaNs
`np.nanmean(a[, axis, dtype, out, keepdims, where])`	Compute the arithmetic mean along the specified axis, ignoring NaNs
`np.nanstd(a[, axis, dtype, out, ddof, ...])`	Compute the standard deviation along the specified axis, while ignoring NaNs
`np.nanvar(a[, axis, dtype, out, ddof, ...])`	Compute the variance along the specified axis, while ignoring NaNs

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.11
NumPy Stastistical Routines: Correlating

Routine	Synopsis
`np.corrcoef(x[, y, rowvar, bias, ddof, dtype])`	Return Pearson product-moment correlation coefficients
`np.correlate(a, v[, mode])`	Cross-correlation of two 1-dimensional sequences
`np.cov(m[, y, rowvar, bias, ddof, fweights, ...])`	Estimate a covariance matrix, given data and weights

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.12
NumPy Statistical Routines: Histograms

Routine	Synopsis
`np.histogram(a[, bins, range, normed, weights, ...])`	Compute the histogram of a dataset
`np.histogram2d(x, y[, bins, range, normed, ...])`	Compute the bi-dimensional histogram of two data samples
`np.histogramdd(sample[, bins, range, normed, ...])`	Compute the multidimensional histogram of some data
`np.bincount(x, /[, weights, minlength])`	Count number of occurrences of each value in array of non-negative ints
`np.histogram_bin_edges(a[, bins, range, weights])`	Function to calculate only the edges of the bins used by the histogram function
`np.digitize(x, bins[, right])`	Return the indices of the bins to which each value in input array belongs

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

So, how do I product a set of summary stats in NumPy?

Snippet 5.13 shows how to create a typical set of summary stats including an array's mean, standard deviation, minimum, maximum, and some percentiles of interest. Mainly, the snippet has three steps. First, we create an array (line 8). Second, we assign a couple of variables to NumPy statistical functions such as `.mean`, `.std`, `.min`, `np.max`, and `.percentile` (see lines 11-17). It is worth noticng that **min** and **max** are keywords reserved for Python builtin functions. To avoid any name conflict and poterntial sources of confusion, in lines 13 and 14, we use the names **min_** and **max_**. Finally, we create a table displaying the variables created in the previous step (see lnes 21, 23, and 27).

Snippet 5.12 — NumPy trigonometric functions in action. Mainly

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # import a module for arranging numbers in a tabular format
5  >>> import tabulate
6
7  # the array
8  >>> X = np.array([0, 0, -3, 12, 7, 2, -4, 6, 9, -1, 5, 3, -1, 3, 10, 9])
9
10 # get the descriptive stats
11 >>> mean = np.mean(X)
12 >>> std = np.std(X)
13 >>> min_ = np.min(X)
14 >>> max_ = np.max(X)
15 >>> pp25 = np.percentile(X, 25)
16 >>> pp50 = np.percentile(X, 50)
17 >>> pp75 = np.percentile(X, 75)
18
19 # arrange the stats in a tabular format
20 # --+ create the table header
21 >>> headers = [
22         "Mean", "St. Dev.", "Min", "Max", "25th pp", "50th pp", "75th pp"
23         ]
24 # --+ format the floating point numbers to two decimal places and get a string
25 >>> stats = [
26         str(np.round(i, 3)) for i in [mean, std, min_, max_, pp25, pp50, pp75]
27         ]
28 # --+ print the table
29 >>> print(tabulate([stats], headers=headers, tablefmt="grid"))
30 +--------+-----------+-------+-------+-----------+-----------+-----------+
31 |   Mean |   St. Dev. |   Min |   Max |   25th pp |   50th pp |   75th pp |
32 +========+===========+=======+=======+===========+===========+===========+
33 |  3.562 |      4.756 |    -4 |    12 |     -0.25 |         3 |       7.5 |
34 +--------+-----------+-------+-------+-----------+-----------+-----------+
```

Can I calaculate Pearson's correlaton coefficients in NumPy?

Yes, you can. Snippet 5.13 shows how to do so by using `.corrcoef`, one of the functions included in Table 5.11.

Snippet 5.13 — a matrix with Pearson's correlation coefficients

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
X = np.array([0, 0, -3, 12, 7, 2, -4, 6, 9, -1, 5, 3, -1, 3, 10, 9])
Y = np.array([12, 12, 4, 3, 9, 2, -6, 15, 0, -12, 15, -3, -1, 0, 0, 1])

# get Pearson's correlation coefficients
>>> np.corrcoef(X, Y)
array([[1.        , 0.19763628],
       [0.19763628, 1.        ]])
```

5.8 Linear Algebra

Is NumPy a good choice when it comes doing linear algebra?

Yes, it is. NumPy offers a rich set of routines, comparable to Matalab's one.[12] Currently, there are five families of routines concerning the broader field linear algebra:

- products
- decomposition
- eigenvalues
- norms
- equations and inversions

For an overview of numPy's linear algebra routines, see Tables 5.13, 5.14, 5.15, 5.16, and 5.17.

TABLE 5.13
NumPy Linear Algebra Routines: Matrix and Vector Products

Routine	Synopsis
`np.dot(a, b[, out])`	Dot product of two arrays
`np.linalg.multi_dot(arrays, *[, out])`	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order
`np.vdot(a, b, /)`	Return the dot product of two vectors
`np.inner(a, b, /)`	Inner product of two arrays
`np.outer(a, b[, out])`	Compute the outer product of two vectors
`np.matmul(x1, x2, /[, out, casting, order, ...])`	Matrix product of two arrays
`np.tensordot(a, b[, axes])`	Compute tensor dot product along specified axes
`np.einsum(subscripts, *operands[, out, dtype, ...])`	Evaluates the Einstein summation convention on the operands
`np.einsum_path(subscripts, *operands[, optimize])`	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays
`np.linalg.matrix_power(a, n)`	Raise a square matrix to the (integer) to the power n
`np.kron(a, b)`	Kronecker product of two arrays

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.14

NumPy Linear Algebra Routines: Decompositions

Routine	Synopsis
`np.linalg.cholesky(a)`	Cholesky decomposition
`np.linalg.qr(a[, mode])`	Compute the qr factorization of a matrix
‘ `np.linalg.svd(a[, full_matrices, compute_uv, ...])`	Singular Value Decomposition

Notes: the statements included in the ‘Routine’ column assume NumPy is loaded with the `np` alias.

TABLE 5.15
NumPy Linear Algebra Routines: Matrix Eigenvalues

Routine	Synopsis
`np.linalg.eig(a)`	Compute the eigenvalues and right eigenvectors of a square array
`np.linalg.eigh(a[, UPLO])`	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix
`np.linalg.eigvals(a)`	Compute the eigenvalues of a general matrix
`np.linalg.eigvalsh(a[, UPLO])`	Compute the eigenvalues of a complex Hermitian or real symmetric matrix

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.16
NumPy Linear Algebra Routines: Norms and Other Number

Routine	Synopsis
`np.linalg.norm(x[, ord, axis, keepdims])`	Matrix or vector norm
`np.linalg.cond(x[, p])`	Compute the condition number of a matrix
`np.linalg.det(a)`	Compute the determinant of an array
`np.linalg.matrix_rank(A[, tol, hermitian])`	Return matrix rank of array using SVD method
`np.linalg.slogdet(a)`	Compute the sign and (natural) logarithm of the determinant of an array
`np.trace(a[, offset, axis1, axis2, dtype, out])`	Return the sum along diagonals of the array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 5.17
NumPy Linear Algebra Routines: Solving Equations and Inverting Matrices

Routine	Synopsis
`np.linalg.solve(a, b)`	Solve a linear matrix equation, or system of linear scalar equations
`np.linalg.tensorsolve(a, b[, axes])`	Solve the tensor equation a x = b for x
`np.linalg.lstsq(a, b[, rcond])`	Return the least-squares solution to a linear matrix equation
`np.linalg.inv(a)`	Compute the (multiplicative) inverse of a matrix
`np.linalg.pinv(a[, rcond, hermitian])`	Compute the (Moore-Penrose) pseudo-inverse of a matrix
`np.linalg.tensorinv(a[, ind])`	Compute the 'inverse' of an N-dimensional array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Matrix and vector products

Snippet 5.14 shows how to use NumPy to get dot, inner, and outer products — whose definitions are reported in equations 5.1, 5.2, and 5.3 respectively:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i \tag{5.1}$$

where x and y are vectors of length n.

$$\langle x, y \rangle = x^T y \tag{5.2}$$

where x^T is the transpose of x.

$$x \otimes y = xy^T \tag{5.3}$$

where y^T is the transpose of y.

Snippet 5.14 — matrix and vector products

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
>>> X = np.arange(1, 11, 1)
>>> X
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

>>> Y = np.arange(10, 0, -1)
>>> Y
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])

# dot product
>>> np.dot(X, Y)
220

# inner product
# --+ let's reshape the arrays
>>> X = X.reshape(5, 2)
>>> X
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
>>> Y = Y.reshape(5, 2)
>>> Y
array([[10,  9],
       [ 8,  7],
       [ 6,  5],
       [ 4,  3],
       [ 2,  1]])
>>> np.inner(X, Y)
array([[ 28,  22,  16,  10,   4],
```

```
       [ 66,  52,  38,  24,  10],
       [104,  82,  60,  38,  16],
       [142, 112,  82,  52,  22],
       [180, 142, 104,  66,  28]])

# outer product
>>> np.outer([0, 1, 2], [4, 5, 6, 7])
array([[ 0,  0,  0,  0],
       [ 4,  5,  6,  7],
       [ 8, 10, 12, 14]])
```

...

Solving a system of equations

Snippet 5.15 shows hot to solve the following system of equations:

$$\begin{cases} 4x + 2y = 2000 \\ 7x + 13y = 3000 \end{cases}$$

Snippet 5.15 — solving a system of equations

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the system of equations
# --+ left-hand side
>>> a = np.array([[4, 2], [7, 13]])
# --+ right-hand side
>>> b = np.array([2000, 3000])

# solve the system of equations
>>> np.linalg.solve(a, b)
array([526.31578947, -52.63157895])
```

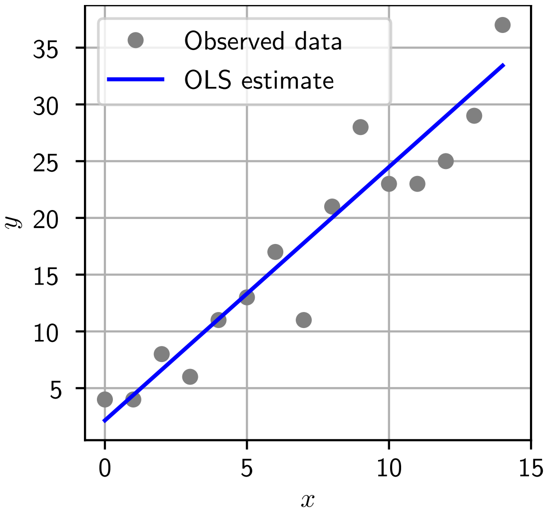
Snippet 5.16 — computing the least-square solutions to a linear matrix equation

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
>>> x = np.arange(0, 15, 1)
>>> y = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

# data preparation / adding a constant to the module
>>> A = np.vstack([x, np.ones(len(X))]).T
>>> A

# estimate the weights (a.k.a., the regression slopes) of the linear model
>>> b = np.linalg.lstsq(A, y)[0]
>>> b
```



5.9 Pseudorandom Number Generation

Snippet 5.15 — a matrix with the eigenvalues and eigenvectors

...

5.10 File Input and Output with `ndarrays`

...

Notes

[8]The Internet has many blog posts showing the performance of NumPy in linear algebra tasks is comparable to compiled languages, such as C.

[9]This feature was introduced with NumPy 1.23.

[10]The documentation of numPy 1.23 states that *It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.*

[11]Here is an interesting passage from the official NumPy documentation: *NumPy hands off array processing to C, where looping and computation are much faster than in Python. To exploit this, programmers using NumPy eliminate Python loops in favor of array-to-array operations. vectorization can refer both to the C offloading and to structuring NumPy code to leverage it.*

[12]Matlab is a numeric computing environment that is particularly popular in academia and industry.

Chapter 6

Data Management with Pandas

...

Chapter 7

Coda

...

Appendices

Appendix A

Cheat Sheets

A.1 Escapes

Table A.1
Helpful Escapes

Escape	Meaning
`\\`	Backslash (stores one \)
`\'`	Single quotes escape (stores ')
`\"`	Double quotes escape (stores ")
`\a`	Bell
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

A.2 String Methods

Table A.2
Comprehensive List of String Methods

Cases I	
s.capitalize()	Capitalize s # 'hello' =>'Hello'
s.lower()	Lowercase s # 'HELLO' =>'hello'
s.swapcase()	Swap cases of all characters in s # 'Hello' =>"hELLO"
s.title()	Titlecase s # 'hello world' =>'Hello World'
s.upper()	Uppercase s # 'hello' =>'HELLO'
Sequence Operations I	
s2 in s	Return true if s contains s2
s + s2	Concat s and s2
len(s)	Length of s
min(s)	Smallest character of s
max(s)	Largest character of s
Sequence Operations II	
s2 not in s	Return true if s does not contain s2
s * integer	Return integer copies of s concatenated # 'hello' =>'hellohellohello'
s[index]	Character at index of s
s[i:j:k]	Slice of s from i to j with step k
s.count(s2)	Count of s2 in s
Whitespace I	
s.center(width)	Center s with blank padding of width # 'hi' =>' hi '
s.isspace()	Return true if s only contains whitespace characters
s.ljust(width)	Left justifiy s with total size of width # 'hello' =>'hello '
s.rjust(width)	Right justify s with total size of width # 'hello' =>' hello'
s.strip()	Remove leading and trailing whitespace from s # ' hello ' =>'hello'
Find / Replace I	
s.index(s2, i, j)	Index of first occurrence of s2 in s after index i and before index j
s.find(s2)	Find and return lowest index of s2 in s
s.index(s2)	Return lowest index of s2 in s (but raise ValueError if not found)
s.replace(s2, s3)	Replace s2 with s3 in s
s.replace(s2, s3, count)	Replace s2 with s3 in s at most count times
s.rfind(s2)	Return highest index of s2 in s
s.rindex(s2)	Return highest index of s2 in s (raise ValueError if not found)
Cases II	
s.casefold()	Casefold s (aggressive lowercasing for caseless matching) # 'ßorat' =>'ssorat'
s.islower()	Return true if s is lowercase
s.istitle()	Return true if s is titlecased # 'Hello World' =>true
s.isupper()	Return true if s is uppercase

TABLE A.2
(Cont'ed)

Inspection I	
s.endswith(s2)	Return true if s ends with s2
s.isalnum()	Return true if s is alphanumeric
s.isalpha()	Return true if s is alphabetic
s.isdecimal()	Return true if s is decimal
s.isnumeric()	Return true if s is numeric
s.startswith(s2)	Return true is s starts with s2
Splitting I	
s.join('123')	Return s joined by iterable '123' # 'hello' =>'1hello2hello3'
s.partition(sep)	Partition string at sep and return 3-tuple with part before, the sep itself, and part after # 'hello' =>('he', 'l', 'lo')
s.rpartition(sep)	Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after # 'hello' =>('hel', 'l', 'o')
s.rsplit(sep, maxsplit)	Return list of s split by sep with rightmost maxsplits performed
s.split(sep, maxsplit)	Return list of s split by sep with leftmost maxsplits performed
s.splitlines()	Return a list of lines in s # 'hello\nworld' =>['hello', 'world']
Inspection II	
s[i:j]	Slice of s from i to j
s.endswith((s1, s2, s3))	Return true if s ends with any of string tuple s1, s2, and s3
s.isdigit()	Return true if s is digit
s.isidentifier()	Return true if s is a valid identifier
s.isprintable()	Return true is s is printable
Whitespace II	
s.center(width, pad)	Center s with padding pad of width # 'hi' =>'padpadhipadpad'
s.expandtabs(integer)	Replace all tabs with spaces of tabsize integer # 'hello\tworld' =>'hello world'
s.lstrip()	Remove leading whitespace from s # ' hello ' =>'hello '
s.rstrip()	Remove trailing whitespace from s # ' hello ' =>' hello'
s.zfill(width)	Left fill s with ASCII '0' digits with total length width # '42' =>'00042'

A.3 NumPy Array Manipulation Routines

TABLE A.3
NumPy Universal Functions: Mathematical Operations

Universal Function	Synopsis
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
matmul(x1, x2, /[, out, casting, order, ...])	Matrix product of two arrays
divide(x1, x2, /[, out, where, casting, ...])	Divide arguments 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$_{d}ivide(x1, x2, /[, out, where, ...])$	Divide arguments element-wise
floor$_{d}ivide(x1, x2, /[, out, where, ...])$	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
float$_{p}ower(x1, x2, /[, out, where, ...])$	First array elements raised to powers from second array, element-wise
remainder(x1, x2, /[, out, where, casting, ...])	Returns the element-wise remainder of division
mod(x1, x2, /[, out, where, casting, order, ...])	Returns the element-wise remainder of division
fmod(x1, x2, /[, out, where, casting, ...])	Returns 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
sqrt(x, /[, out, where, casting, order, ...])	Return the non-negative square-root of an array, element-wise

TABLE A.4
NumPy Universal Functions: Trigonometric Operations

Universal Function	Synopsis
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
degrees(x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees
radians(x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians
deg2rad(x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians
rad2deg(x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees

Table A.5
NumPy Universal Functions: Floating Operations

Universal Function	Synopsis
isfinite(x, /[, out, where, casting, order, ...])	Test element-wise for finiteness (not infinity and 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, ...])	Returns 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

Table A.6
NumPy Universal Functions: Comparison Operations

Universal Function	Synopsis
greater(x1, x2, /[, out, where, casting, ...])	Return the truth value of (x1 ¿ x2) element-wise
greater$_{e}qual(x1, x2, /[, out, where, ...])$	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$_{e}qual(x1, x2, /[, out, where, casting, ...])$	Return the truth value of (x1 ¡= x2) element-wise
not$_{e}qual(x1, x2, /[, out, where, casting, ...])$	Return (x1 != x2) element-wise
equal(x1, x2, /[, out, where, casting, ...])	Return (x1 == x2) element-wise
logical$_{a}nd(x1, x2, /[, out, where, ...])$	Compute the truth value of x1 AND x2 element-wise
logical$_{o}r(x1, x2, /[, out, where, casting, ...])$	Compute the truth value of x1 OR x2 element-wise
logical$_{x}or(x1, x2, /[, out, where, ...])$	Compute the truth value of x1 XOR x2, element-wise
logical$_{n}ot(x, /[, out, where, casting, ...])$	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

Table A.7
NumPy Array Manipulation Routines

Routine	Synopsis
Basic operations	
`np.copyto(dst, src[, casting, where])`	Copies values from one array to another, broadcasting as necessary
`np.shape(a)`	Return the shape of an array
Changing array shape	
`np.reshape(a, newshape[, order])`	Gives a new shape to an array without changing its data
`np.ravel(a[, order])`	Return a contiguous flattened array
`np.ndarray.flat`	A 1-D iterator over the array
`np.ndarray.flatten([order])`	Return a copy of the array collapsed into one dimension
Changing array shape	
`np.moveaxis(a, source, destination)`	Move axes of an array to new position
`np.rollaxis(a, axis[, start])`	Roll the specified axis backwards, until it lies in a given position
`np.swapaxes(a, axis1, axis2)`	Interchange two axes of an array
`np.ndarray.T`	The transposed array
`np.transpose(a[, axes]`	Reverse or permute the axes of an array; returns the modified array
Changing number of dimensions	
`np.atleast_1d(*arys)`	Convert inputs to arrays with at least one dimension
`np.atleast_2d(*arys)`	View inputs as arrays with at least two dimensions
`np.atleast_3d(*arys)`	View inputs as arrays with at least three dimensions
`np.broadcast`	Produce an object that mimics broadcasting
`np.broadcast_to(array, shape[, subok])`	Broadcast an array to a new shape
`np.broadcast_arrays(*args[, subok])`	Broadcast any number of arrays against each other
`np.expand_dims(a, axis)`	Expand the shape of an array
`np.squeeze(a[, axis])`	Remove axes of length one from a

TABLE A.7 (cont'd)

Changing kind of array	
`np.asarray(a[, dtype, order, like])`	Convert the input to an array
`np.asanyarray(a[, dtype, order, like])`	Convert the input to an ndarray, but pass ndarray subclasses through
`np.asmatrix(data[, dtype])`	Interpret the input as a matrix
`np.asfarray(a[, dtype])`	Return an array converted to a float type
`np.asfortranarray(a[, dtype, like])`	Return an array (ndim >= 1) laid out in Fortran order in memory
`np.ascontiguousarray(a[, dtype, like])`	Return a contiguous array (ndim >= 1) in memory (C order)
`np.asarray_chkfinite(a[, dtype, order])`	Convert the input to an array, checking for NaNs or Infs
`np.require(a[, dtype, requirements, like])`	Return an ndarray of the provided type that satisfies requirements
Joining arrays	
`np.concatenate([axis, out, dtype, casting])`	Join a sequence of arrays along an existing axis
`np.stack(arrays[, axis, out])`	Join a sequence of arrays along a new axis
`np.block(arrays)`	Assemble an nd-array from nested lists of blocks
`np.vstack(tup)`	Stack arrays in sequence vertically (row wise)
`np.hstack(tup)`	Stack arrays in sequence horizontally (column wise)
`np.dstack(tup)`	Stack arrays in sequence depth wise (along third axis)
`np.column_stack(tup)`	Stack 1-D arrays as columns into a 2-D array
`np.row_stack(tup)`	Stack arrays in sequence vertically (row wise)
Splitting arrays	
`np.split(ary, indices_or_sections[, axis])`	Split an array into multiple sub-arrays as views into array
`np.array_split(ary, indices_or_sections[, axis])`	Split an array into multiple sub-arrays
`np.dsplit(ary, indices_or_sections)`	Split array into multiple sub-arrays along the 3rd axis (depth)
`np.hsplit(ary, indices_or_sections)`	Split an array into multiple sub-arrays horizontally (column-wise)
`np.vsplit(ary, indices_or_sections)`	Split an array into multiple sub-arrays vertically (row-wise)

Table A.7 (cont'd)

Tiling elements	
`np.tile(A, reps)`	Construct an array by repeating A the number of times given by reps
`np.repeat(a, repeats[, axis])`	Repeat elements of an array
Adding and removing elements	
`np.delete(arr, obj[, axis])`	Return a new array with sub-arrays along an axis deleted
`np.insert(arr, obj, values[, axis])`	Insert values along the given axis before the given indices
`np.append(arr, values[, axis])`	Append values to the end of an array
`np.resize(a, new_shape)`	Return a new array with the specified shape
`np.trim_zeros(filt[, trim])`	Trim the leading and/or trailing zeros from a 1-D array or sequence
`np.unique(ar[, return_index, return_inverse, ...])`	Find the unique elements of an array
Rerraning elements	
`np.flip(m[, axis])`	Reverse the order of elements in an array along the given axis
`np.fliplr(m)`	Reverse the order of elements along axis 1 (left/right)
`np.flipud(m)`	Reverse the order of elements along axis 0 (up/down)
`np.reshape(a, newshape[, order])`	Gives a new shape to an array without changing its data
`np.roll(a, shift[, axis])`	Roll array elements along a given axis
`np.rot90(m[, k, axes])`	Rotate an array by 90 degrees in the plane specified by axes

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Appendix B

Collaborative and Versioning Tools

...