PYTHON3.3

Hao Zhang
zhangh0214@gmail.com

Contents

List of Tables

I PYTHON BASIS

1 Getting Started

1.1 Python2 or Python3

Python3 is now seen as a sandbox for exploring new ideas, while Python2 is viewed as the tried-and-true Python, which does not have all of Python3's features but is still more pervasive.

1.2 How Python Runs Programs

1.2.1 Introducing the Python Interpreter

DEFINITION 1.1 **Interpreter** A kind of program that executes other programs.

Remark 1.1 When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

1.2.2 Program Execution

The programmer's view. A Python program is a text file containing Python statement, and you run those files through the interpreter.
Python's view. Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.

DEFINITION 1.2 **Byte Code** A lower-level, platform-independent Python-specific representation (not binary machine code) of your source code. Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution, i.e., byte code can be run much more quickly than the original source code.

Remark 1.2 When files are imported, Python will write the byte code of your programs in a `.pyc` file with name identifying the Python version that created them (e.g., `script.cpython-33.pyc`), and store it in the `__pycache__` subdirectory. The next time you run your program, Python will load the `.pyc` files and skip the compilation step, as long as you have not changed your source code since last saved.

DEFINITION 1.3 **Python Virtual Machine (PVM)** A big code loop that iterates through your byte code instructions, one by one, to carry out their operations. This is called the Python interpreter.

1.2.3 Execution Model

CPython is the standard implementation, and it is coded in portable ANSI C language code. It implements the Python language by compiling source code to byte code and executing the byte code on an appropriate virtual machine.

1.2.4 Execution Optimization Tools

The Cython system is a hybrid language that combines Python code with the ability to call C functions and use C type declarations for variables, parameters, and class attributes. Cython code can be compiled to C code that uses the Python/C API, which may then be compiled completely. Though not completely compatible with standard Python, Cython can be useful both for wrapping external C libraries and for coding efficient C extensions for Python.

1.3 How You Run Programs

1.3.1 The Interactive Prompt

1.3.2 System Command Lines and Files

DEFINITION 1.4 **Module** A text file containing Python statements.

DEFINITION 1.5 **Program** A series precoded statements stored in a file for repeated execution. Module files are often referred to as programs in Python.

DEFINITION 1.6 **Script** A main or top-level program file, i.e., a file launched to start the entire program. We usually reserve the term "module" for a file imported from another file, and "script" for the main file of a program.

Python's `-i` command-line argument. In this case, Python will enter into its interactive interpreter mode when your script exits, whether it ends successfully or runs into an error. At this point, you can print the final values of variables to get more details about what happened in your code because they are in the top-level namespace. You can also then import and run the pdb debugger for even more context.
Python's `-O` command-line argument. In this case, Python will run in optimized mode.

1.3.3 Unix-Style Executable Scripts: `#!`

You can also turn files of Python code into executable programs. Unix-style executable scripts are just normal text files containing Python statements, but with two special properties:

- Their first line is special. Scripts usually start with a line that begins with the characters `#!` (often called "hash bang" or "shebang"), followed by the path to the Python interpreter on your machine, e.g., `#!/usr/bin/env python`. The `env` program locates the Python interpreter according to your system search path according to `PATH`. This scheme can be more portable, as you do not need to hardcode a Python install path in the first line of all your scripts.
- They usually have executable privileges. Script files are usually marked as executable to tell the operating system that they may be run as top-level programs. On Unix systems, a command such as `chmod +x file.py` usually does the trick.

1.3.4 Souce Code Encoding

By default, Python source files are treated as encoded in UTF-8. To declare an encoding other than the default one, a special comment line should be added as the first line. One exception is when the souce code starts with a Unix "hash bang" line. In this case, the encoding declaration should be added as the second line.

```
1 # -*- coding: utf-8 -*-
```

1.3.5 Module Imports and Reloads

DEFINITION 1.7 **Import** An operation to load another file and grant access to that file's contents. Import operations execute the code in a file that is being loaded as a last step.

Module as a namespace. A module can be regarded as a package of variable names, known as a namespace, and the names within that package are called attributes. Importers gain access to all the names assigned at the top level of a module's file, like functions, classes, variables, etc. You can use `dir(mod)` to fetch a list of all the names available inside a module.

DEFINITION 1.8 **Attribute** A variable name that is attached to a specific object, e.g., a module.

Access attributes. There are two different ways.

- You can load the module as a whole with an `import` statement, and then qualify the module name with the attribute name to fetch it, e.g., `mod.name`.
- You can fetch (really, copy) names out of a module with `from` statements, such that they become variables in the current namespace, so we can use names directly.

Import and reload. Since imports are too expensive an operation which must find files, compile them to byte code, and run the code, after the first import, later imports do nothing. If you really want to force Python to run the file again, you need to call the `reload()`

function. Note that reloads are not transitive, i.e., reloading a module reloads that module only, not any modules it may import.

```
import imp
# Load and run the current version of your file's code.
imp.reload(file)
```

However, names loaded with a `from` are not directly updated by a `reload()`, but names accessed with an `import` statement are.

2 Types and Operations

2.1 Introducing Python Object Types

2.1.1 The Python Conceptual Hierarchy

Python programs can be decomposed into modules, statements, expressions, and objects, as follows.

- Expressions create and process objects.
- Statements contain expressions.
- Modules contain statements.
- Programs are composed of modules.

DEFINITION 2.1 **Object** A piece of memory, with values and sets of associated operations. Everything is an object in a Python script.

DEFINITION 2.2 **Expression** A combination of numbers (or other objects) and operators that computes a value when executed by Python.

DEFINITION 2.3 **Statement** The smallest unit you write to tell Python what your programs should do. To make a single statement span across multiple lines, you have to enclose part of your statement in a bracketed pair, either (), [], or {}. Any code enclosed in these constructs can cross multiple lines. Your statement does not end until Python reaches the line containing the closing part of the pair.

2.1.2 Why Use Built-in Types?

In contrast to lower-level languages such as C or C++, Python provides powerful object types as an intrinsic part of the language.

2.1.3 Python's Core Data Types

Table 2.1 previews Python's built-in object types.

DEFINITION 2.4 **Literal** An expression whose syntax generates an object.

`True` and `False` in Python. In Python, an integer `0` represents `False`, and an integer `1` represents `True`. Python recognizes any empty data structure as `False` and any nonempty data structure as `True`.

The `None` object. `None` is a special data type in Python and typically serves as an empty placeholder (much like `nullptr` in C++). `None` is considered to be `False`.

The `type` object. The type of an object is an object of type `type`. A call to the built-in function `type(x)` returns the type object of object `x`.

Table 2.1
Built-in objects preview.

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))`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Sets	`set('abc'), {'a', 'b', 'c'}`
Files	`open('eggs.txt'), open('ham.bin', 'wb')`
Other core types	Booleans, `None`, `type`
Program unit types	Functions, modules, classes

Table 2.2
Numeric literals and constructors. Floating-point numbers are implemented as C `doubles` in standard CPython. `True` and `False` behave exactly like the integers 1 and 0 in arithmetic operations.

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

2.2 Numeric Types

2.2.1 Numeric Type Basics

A complete inventory of Pythons numeric literals is illustrated in Table 2.2.

2.2.2 Built-in functions and modules

Built-in functions for numeric types are summarized in Table 2.3, and built-in modules for numeric types are summarized in Table 2.4. Another common built-in functions are summarized in Table 2.5

`str()` vs. `repr()`. `repr()` (and the default interactive echo) produces results that look as though they were code, while `str()` (and the `print` operation) converts to a typically more user-friendly format if available.

Table 2.3
Built-in functions for numeric types.

Function	Result
Mathematical Operations	
`abs(x)`	Return the absolute value of a number.
`divmod(a, b)`	Return the quotient and remainder `(a//b, a%b)`.
`max(iterale[, key])`	Return the largest item in an iterable. The optional `key` argument specifies a one-argument ordering function like that used for `list.sort()`. The same as the following.
`max(arg1, arg2, *args[, key])`	Return the largest of two or more arguments.
`min(iterale[, key])`	Return the smallest item in an iterable.
`min(arg1, arg2, *args[, key])`	Return the smallest of two or more arguments.
`pow(x, y[, z])`	Return `x**y`. If `z` is present, return `(x**y)%z`.
`round(number, ndigits=0)`	Return `number` rounded to `ndigits` digits.
`sum(iterable, start=0)`	Return the sum of `start` and the elements of `iterable` from left to right.
Type Conversions	
`bin(x)`	Convert an integer number to a binary string.
`hex(x)`	Convert an integer number to a lowercase hexadecimal string prefixed with `0x`.
`int(x, base=10)`	Convert a string to an integer using a base `base`.
`chr(i)`	Return a string of one character whose ASCII code is `i`. `i` should be in range [0, 255].
`ord(c)`	Given a string of length one, return an integer representing the Unicode code/value of the byte of the character when `c` is a Unicode object/8-bit string.

Table 2.4
Built-in modules for numeric types.

Function	Result
`math`	
`math.e`	`2.718281`
`math.pi`	`3.141592`
`math.floor(x)`	Return the floor of `x`.
`math.trunc()`	Return the real value `x` truncated to an integer towards zero.
`random.choice(seq)`	Return a random element from the non-empty sequence `seq`.
`random.gauss(mu, sigma)`	Return a random floating point number in $\mathcal{N}$(`mu`, `sigma`^{2}).
`random.random()`	Return a random floating point number in [0.0, 1.0).
`random.randrange(start=0, stop, step=1)`	Return a randomly selected integer from [`start`, `stop`)
`random.sample(seq, k)`	Return a length-`k` list of random sampling without replacement from `seq`
`random.seed(a=None)`	Initialize the random number generator by `a`. If omitted, the system time will be used.
`random.shuffle(seq)`	Shuffle the sequence `seq` in place.
`random.uniform(a, b)`	Return a random floating point number in [a, b].

Table 2.5
Other common Python built-in functions.

Function	Result
Logic Tests	
`all(iterable)`	Return `True` if `iterable` is empty or all elements of the `iterable` is `True`.
`any(iterable)`	Return `True` if `iterable` is not empty and if any element of the `iterable` is `True`.
Python Interpreter	
`dir([object])`	Return the list of names in the current local scope. With an argument, return a list of valid attributes for that object. `dir(object)` behaves differently with different types of objects. If `object` is a module object, the list contains the names of the module's attributes. If `object` is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
`help([object])`	Invoke the built-in help system. If `object` is given, a help page on `object` is generated.
IO	
`print(*objects, sep=' ', end='\n', file=sys.stdout)`	Print `objects` to the stream `file`, separated by `sep` and followed by `end`. All non-keyword arguments are converted to strings like `str()` does.
`raw_input([prompt])`	Read a line from input, converts it to a string (stripping a trailing newline), and return that. If `prompt` is present, it is written to standard output without a trailing newline.
Object Operations	
`delattr(object, name)`	Delete the named attribute `name`, which is a string type.
`hasattr(object, name)`	Return `True` if `name` is one of `object`'s attributes.
`setattr(object, name, value)`	Assign `value` to the attribute `name`, which may name an existing attribute or a new attribute of `object`.
`hash(object)`	Return the hash value of `object`.
`id(object)`	Return the address of the object in memory, which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.
`callable(object)`	Return `True` if `object` is callable.
`isinstance(object, class-info)`	Return `True` if `object` is an instance of `classinfo`. If `classinfo` is a tuple, return `True` if `object` is an instance of any of the classes or types.

main 2017/11/23 21:35 Page 11 #19

2.3 The Dynamic Typing Interlude

2.3.1 The Case of the Missing Declaration Statements

Python is dynamically typed and strongly typed.

DEFINITION 2.5 **Dynamically Typed** A model that determines the type for you automatically at runtime, instead of requiring declarations in your code.

DEFINITION 2.6 **Strongly Typed** A constraint that means you can perform on an object only operations that are valid for its type.

In Python, variables are created when assigned, can reference any type of object, and must be assigned before they are referenced. Variables and objects are stored in different parts of memory. The variable is really a `void *` pointer to the object's memory space. The link (i.e., pointer) is called reference in Python.

- **Variable creation.** A variable is created when your code first assigns it a value. Future assignments change the value of the already created name.
- **Variable types.** A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names.
- **Variable use.** When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be.

Objects have more structure than just enough space to represent their values. Each object also has two standard header fields: a type designator used to mark the type of the object, and a reference counter used to determine when it is OK to reclaim the object (i.e., garbage collection).

2.3.2 Shared References

DEFINITION 2.7 **Shared Reference** Situation when multiple names (i.e., variables) referencing the same object.

Shallow vs. deep copy. The standard library `copy` module has a call for copying any object type generically, as well as a call for copying nested object structures.

```
1 import copy
2 y = copy.copy(x)       # Top-level shallow copy
3 y = copy.deepcopy(x)   # Copy all nested parts
```

`==` vs. `is`. The == operator tests whether the two referenced objects have the same values, and it is useful for equality checks. On the other hand, the `is` operator tests for object identity, i.e., it returns `True` only if both names point to the exact same object. In fact, `is`

simply compares the pointers that implement references, and it serves as a way to detect shared references in your code if needed.

As a side note, small integers and strings are cached and reused in Python. Since you cannot change immutable numbers or strings in place, it does not matter how many references there are to the same object. Every reference will always see the same, unchanging value.

2.3.3 Dynamic Typing Is Everywhere

Dynamic typing is also the root of Python's polymorphism.

2.4 Sequence Types

DEFINITION 2.8 **Python Sequence** 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. There are 7 sequence types in Python: `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, and `xrange`.

Most sequence types support the operations in Table 2.6, and they are ordered in ascending priority.

Table 2.6
Operations supported by most sequence types, where `s` and `t` are sequences of the same type; `n`, `i`, and `j` are arrays.

Operation	Result
`x in s`	`True` if an item of `s` is equal to `x`. When `s` is a string or Unicode string object, the `in` and `not in` operations act like subtring test. `x` may be a string of any length.
`x not in s`	`False` if an item of `s` is equal to `x`.
Comparasions	Lists and tuples are compared lexicographically by comparing corresponding elements.
`s + t`	The concatenation of `s` and `t`. If `s` and `t` are both strings, it is prefer to use `str.join()` method instead of string concatenation such as `s=s+t` or `s+=t`. The former assures linear time performance, while the latter has quadratic running time.
`s * n, n * s`	Adding `s` to itself `n` times. Values of `n` less than 0 are treated as 0 (which yields an empty sequence of the same type as `s`). Note that items in the sequence `s` are not copied; they are referenced multiple times.
`s[i]`	`i`-th item of `s`, origin 0.
`s[i=0:j=len(s)]`	Slice of `s` from `i` to `j`, which is defined as [`i`, `j`). If `i` or `j` is > `len(s)`, use `len(s)`. If $i \geq j$, the slice is empty. If `i` or `j` is negative, the index is relative to the end of sequence `s`: `len(s)+i` or `len(s)+j` is substituted. But note that `-0` is still `0`. For nonnegative indices, the length of a slice is `j-i`, if both are within bounds.
`s[i:j:k=1]`	Slice of `s` from `i` to `j` with step `k`, which is defined as $\{i + nk \mid 0 \leq n < (j - i)/k\}$. In other words, it stops when `j` is reached (but never including `j`). When `k` > 0, `i` and `j` are reduced to `len(s)` if they are greater; when `k` < 0, `i` and `j` are reduced to `len(s)-1` if they are greater. If `i` or `j` is omitted or `None`, they become "end"values (which end depends on the sign of `k`). `k` cannot = 0.
`len(s)`	Length of `s`.
`min(s)`	Smallest item of `s`.
`max(s)`	Largest item of `s`.

Table 2.7
Common string literals.

Object type	Example literals
Empty string	`''`
Normal strings	`"spam's", 's\np\ta'`
Triple-quoted block strings	`"""...multiline..."""`
Raw strings	`r'\temp\spam'`
Byte strings	`b'sp\xc4m'`
Unicode strings	`u'sp\u00c4m'`

2.5 String Fundamentals

2.5.1 Unicode

ASCII is a simple form of Unicode text. In Python3, `str` is used for Unicode text, and `bytes` is used for binary data (including encoded text). Files work in two modes: text which represents content as `str` and implements Unicode encodings, and binary which deals in raw `bytes` and does no data translation.

2.5.2 String Basics

Table 2.7 previews common string literals. Beside, Python automatically concatenates adjacent string literals in any expression.

2.5.3 String Methods

String methods are summarized in Table 2.8, 2.9, and 2.10.

Table 2.8
String methods. The original string is returned if `width` <= the string length.

Method	Result
Checking	
`str.isalnum()`	`True` if there is ≥ 1 character, and all characters are alphanumeric.
`str.isalpha()`	`True` if there is ≥ 1 character, and all characters are alphabetic.
`str.isdigit()`	`True` if there is ≥ 1 character, and all characters are digits.
`str.isspace()`	`True` if there is ≥ 1 character, and all characters are whitespace characters.
`str.islower()`	`True` if there is ≥ 1 character, and all cased characters are lowercase.
`str.isupper()`	`True` if there is ≥ 1 character, and all cased characters are uppercase.
`str.istitle()`	`True` if there is ≥ 1 character, and it is a titlecased string. Titlecased string means that uppercase characters may only follow uncased characters and lowercase characters only follow cased ones.
`str.startswith(prefix[, start[, end]])`	`True` if string starts with `prefix`, which can be a tuple of prefixes to look for. With optional `start`, test string beginning at that position; with optional `end`, test string stopping at that position.
`str.endswith(suffix[, start[, end]])`	`True` if string ends with `suffix`, which can be a tuple of suffixes to look for.
Transformation	
`str.lower()`	Return a copy of the string with all the cased characters converted to lowercase.
`str.upper()`	Return a copy of the string with all the cased characters converted to uppercase.
`str.swapcase()`	Return a copy of the string with uppercase characters converted to lowercase and vice versa.
`str.capitalize()`	Return a copy of the string with its first characater capitalized and the rest lowercased.
`str.title()`	Return a copy of the string where words start with an uppercase character and the remaining characters are lowercase.
`str.expandtabs(tabsize=8)`	Return a copy of the string where all tab characters are replaced by spaces.
Padding	
`str.center(width, fillchar=' ')`	Return the centered string of length `width`.
`str.ljust(width, fillchar=' ')`	Return the left justified string of length `width`. The original string is returned if `width` <= the string length.
`str.rjust(width, fillchar=' ')`	Return the right justified string of length `width`.
`str.zfill(width)`	Return the string of length `width` left filled with `0`'s.

Table 2.9

String methods (continued). `chars` is a string specifying the set of characters to be removed, and it defaults to removing whitespace. With optional `start`, test string beginning at that position; with optional `end`, test string stopping at that position.

Methods	Results
Strip	
`str.strip([chars])`	Return a copy of the string with leading and trailing characters removed.
`str.lstrip([chars])`	Return a copy of the string with leading characters removed.
`str.rstrip([chars])`	Return a copy of the string with trailing characters removed.
Searching	
`str.count(sub[, start[, end]])`	Return the number of non-overlapping occurrences of substring `sub`.
`str.find(sub[, star[, end]])`	Return the lowest index in the string where substring `sub` is found; return `-1` if not found.
`str.index(sub[, star[, end]])`	Return the lowest index in the string where substring `sub` is found; raise `ValueError` if not found.
`str.rfind(sub[, star[, end]])`	Return the highest index in the string where substring `sub` is found; return `-1` if not found.
`str.rindex(sub[, star[, end]])`	Return the highest index in the string where substring `sub` is found; raise `ValueError` if not found.
Replace	
`str.replace(old, new[, count])`	Return a copy of the string with all non-overlapping occurrence of string `old` replaced by `new`. If `count` is given, only the first `count` occurrences are replaced.
`str.translate(table[, deletechars])`	Return a copy of the string where each character is mapped through the given translation table. If `deletechars` is given in method `str.translate()`, all characters occurring in `deletechars` are removed. `table` must be a string of length 256. `string.maketrans(from, to)` returns a table suitable for `str.translate()`, that will map each character in `from` into the character at the same position in `to`; `from` and `to` must have the same length. `table` can be set to `None` if you only want to delete characters.

Table 2.10
String methods (continued).

Methods	Results
Split	
`str.partition(sep)`	Split the string at the first occurrence of the string `sep`. `str.partition()` return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.
`str.rpartition(sep)`	Split the string at the last occurrence of the string `sep`.
`str.split([sep[, maxsplit]])`	Return a list of the words in the string, using the string `sep` as the delimiter string. If `maxsplit` is given, $\leq$ `maxsplit` splits are done (thus, the list will have $\leq$ `maxsplit+1` elements). If `sep` is given, consecutive delimiters are deemed to delimit empty strings (e.g., `'1,,2'.split(',')` returns `['1', '', '2']`). If `sep` is omitted or `None`, consecutive whitespace are regarded as a single speparater, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace (e.g., `' 1 2 3 '.split(None)` returns ['1', '2', '3']).
`str.rsplit([sep[, maxsplit]])`	Return a list of the words in the string, using the string `sep` as the delimiter string.
`str.splitlines(keepends=False)`	Return a list of the lines in the string, breaking at `\r`, `\n`, or `\r\n`. Those line breaks are not included in the resulting string unless `keepends` is `True`.
Concatenation	
`str.join(iterable)`	Return a string which is the concatenation of the strings in `iterable`. The separator between elements is the string providing this method.
Encoding	
`str.encode([encoding, errors='strict'])`	Return an encoded version of the string. The default encoding is the current default string encoding.
`str.decode([encoding, errors='strict'])`	Return an decoded version of the string. The default encoding is the current default string encoding.

Table 2.11
String conversion flag characters.

Flag	Meaning
`0`	Zero padding for numeric values.
`-`	Left justification.
`+`	Numeric sign character.
`m.n`	Put the value in a field with total width `m`, and `n` characters to the right of the decimal point.

Table 2.12
String conversion type characters. The precision determines the number of digits after the decimal point defaults to 6.

Conversion	Meaning
`d`	Signed integer decimal.
`x`	Signed hexadecimal (lowercase).
`X`	Signed hexadecimal (uppercase).
`e`	Floating point exponential format (lowercase).
`f`	Floating point decimal format.
`c`	Single character (accepts integer or single character string).
`s`	String (converts any Python object using `str()`).
`%`	A `%` character.

2.5.4 String Formatting Expressions

Given `format % values`, where `format` is a string or Unicode object, `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to the using `sprintf()` in the C language. The conversion flag characters are summarized in Table 2.11, and the conversion types are summarized in Table 2.12.

If `format` requires a single argument, `values` may be a single non-tuple object. Otherwise, `values` must be a tuple with exactly the number of items specified by the `format` string, or a single mapping object (e.g., a dictionary). When `values` is a mapping type, `format` in the string must include a parenthesised mapping key into that dictionary inserted immediately after the `%` character.

```
'%(language)s has %(number)03 quote types.' % \
    {'language': Python, 'number': 2}
```

2.6 Lists and Tuples

2.6.1 List Methods

Table 2.13 summarizes common list methods.

Table 2.13
List operations and methods.

Operation	Result
Updating	
`list[i] = x`	Item `i` of the list is replaced by `x`.
`list[i:j] = t`	Slice of the list `i:j` is replaced by the contents of the iterable `t`.
`list[i:j:k] = t`	Slice of the list `i:j:k` is is replaced by the contents of the iterable `t`. `t` must have the same length as the slice it is replacing.
Removing	
`del list[i]`	Same as `list[i] = []`.
`del list[i:j]`	Same as `list[i:j] = []`.
`del list[i:j:k]`	Same as `list[i:j:k] = []`.
`del list`	Clear the entire list. Reference the name hereafter is an error.
`list.pop(i=-1)`	Same as `x = list[i]; del list[i]; return x`. By default, the last item is removed and returned.
`list.remove(x)`	Same as `del list[list.index(x)]`.
Inserting	
`list.append(x)`	Same as `list[len(list):len(list)] = [x]`. Unlike + concatenation, `append()` does not have to generate new objects, so it is usually faster than +.
`list.insert(i, x)`	Same as `list[i:i] = [x]`.
`list.extend(t)`	Same as `list[len(list):len(list)] = t`.
`list += t`	Same as `list[len(list):len(list)] = t`.
`list *= n`	Update `list` with its contents repeated `n` times.
Searching	
`list.count(x)`	Return number of `i`'s for which `list[i] == x`.
`list.index(x[, i[, j]])`	Return smallest `i <= k <j` such that `list[k] == x`.
Reversing/Sorting	
`list.reverse()`	Reverse the items of `list` in place.
`list.sort([key, reverse=False])`	Sort the items of `list` in place. `key` specifies a function of one argument that is used to extract a comparison key from each list element.

2.6.2 Tuples

Tuples work exactly like lists, except that tuples cannot be changed in place (they are immutable). Empty tuples are constructed by an empty pair of parentheses, e.g., `()`, and a tuple with one item is constructed by following a value with a comma, e.g., `('hello',)`.

2.7 Dictionaries

2.7.1 Dictionary Methods

Table 2.14 summarizes common dictionary methods.

Table 2.14
Dictionary operations and methods.

Operation	Result
Membership Testing	
`key in dict`	Return `true` if `dict` has a key `key`.
`key not in dict`	Return `true` if `dict` does not have a key `key`.
Accessing	
`len(dict)`	Return the number of items in `dict`.
`dict[key]`	Return the item of `dict` with key `key`. raise `keyerror` if `key` is not in the map.
`dict.get(key, default=none)`	Return the value of `key` if `key` is in `dict`, else `default`.
`dict.keys()`	Return a copy of `dict`'s list of keys.
`dict.values()`	Return a copy of `dict`'s list of values.
`dict.items()`	Return a copy of `dict`'s list of `(key, value)` pairs.
`dict.iterkeys()`	Return an iterator over `dict`'s list of keys. it cannot be used to adding/deleting entries.
`dict.itervalues()`	Return an iterator over `dict`'s list of values. it cannot be used to adding/deleting entries.
`dict.iteritems()`	Return an iterator over the dictionary's `(key, value)` pairs. it cannot be used to adding/deleting entries.
`dict.viewkeys()`	Return a set-like object providing a view on `dict`'s keys.
`dict.viewkeys()`	Return a set-like object providing a view on `dict`'s values.
`dict.viewitems()`	Return a set-like object providing a view on `dict`'s `(key, value)` pairs.
Updating	
`dict[key] = value`	Set `dict[key]` to value.
`dict.setdefault(key, default=none)`	If `key` is in `dict`, return its value. if not, insert `key` with a value of `default` and return `default`.
`dict.update(other)`	Update `dict` with key-value pairs from `other`, overwriting existing keys.
Removing	
`del dict[key]`	Remove `dict[key]` from `dict`. Raise `keyerror` if `key` is not in the map.
`dict.clear()`	Remove all items from `dict`.
`dict.pop(key, default)`	If `key` is in `dict`. remove it and return its value, else return `default`.
`dict.popitem()`	Remove and return an arbitrary `(key, value)` pair from `dict`.
Copying/Creating	
`dict.copy()`	Return a shallow copy of `dict`.
`dict.fromkeys(seq, value=none)`	Create a new dictionary with keys from `seq` and values set to `value`.

2.8 Sets

2.8.1 Set Methods

Table 2.15 summarizes common set methods.

Table 2.15
Set operations and methods. `other` should also be a set.

Operation	Result
Mathematical Set Operations	
`len(set)`	Return the number of elements in `set`.
`x in set`	Test `x` for membership in `set`.
`x not in set`	Test `x` for non-membership in `set`.
`set.isdisjoint(other)`	Return `True` if `set` has no elements in common with `other`. Sets are disjoint iff their intersection is the empty set.
`set <= other`	Test whether every element in `set` is in `other`.
`set < other`	Test whether `set` is a proper subset of `other`.
`set >= other`	Test whether every element in `other` is in `set`.
`set > other`	Test whether `set` is a proper superset of `other`.
`set | other | ...`	Return a new set with elements from `set` and `other`, etc.
`set |= other | ...`	Update the set, adding elements from all others.
`set & other & ...`	Return a new set with elements common to `set` and `other`, etc.
`set &= other & ...`	Update the set, keeping only elements found in it and all others.
`set - other`	Return a new set with elements in `set` that are not in `other`.
`set -= other | ...`	Update the set, removing elements found in others.
`set ^ other`	Return a new set with elements in either `set` or `other` but not both.
`set ^= other`	Update the set, keeping only elements found in either set, but not both.
Updating	
`set.copy()`	Return a new set with shallow copy of `set`.
`set.add(elem)`	Add `elem` to the set.
`set.remove(elem)`	Remove `elem` from the set. Raise `KeyError` if `elem` is not contained in the set.
`set.discard(elem)`	Remove `elem` from the set. Do nothing if `elem` is not contained in the set.
`set.pop()`	Remove and return an arbitrary element from the set. Raise `KeyError` if the set is empty.
`set.clear()`	Remove all elements from the set.

2.9 Files

2.9.1 File Methods

Table 2.16 summarizes common file methods.

Table 2.16
File-related methods.

Method	Result
`open(name, mode='r')`	Open a file and return the `file` object. `open()` accepts a file path in which directories and files are separated by `/`, regardless of the proclivities of the underlying operating systems. If the file path argrument does not include the file's directory name, the file is assumed to reside in the current working directory. `mode` can be `'r'`, `'w'`, and `'a'`. `'r+'` opens the file for both reading and writing. `'rU'` can read file independently of the line-termination convention the files are using for different operating systems.
`file.close()`	Close the file. You can avoid having to call `file.close()` explicitly if you use the `with` statement. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.
`file.tell()`	Return the file's current position, measured in bytes from the beginning of the file.
`file.seek(offset, whence=os.SEEK_SET)`	Set the file's current position. `whence` defaults to mean absolute file positioning (seek relative to the file's beginning). Other values are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the file's end).
`file.read([size])`	Read $\leq$ `size` bytes from the file and return it as a string. If `size` if omitted, read all data until EOF is reached.
`file.readline()`	Read one entire line from the file and return it as a string. A trailing newline character `\n` is kept in the string.
`file.readlines()`	Read until EOF using `file. readline()` and return a list containing the lines thus read.
`file.write(str)`	Write a string to the file.
`file.writelines(sequence)`	Write a sequence of strings (typically a list of strings) to the file.

Table 2.17
Common `pickle` methods.

Operation	Result
`pickle.dump(obj, file)`	Write a pickled representation of `obj` to the open file object `file`.
`pickle.load(file)`	Read a pickled object representation from the open file object `file` and return the reconstituted object hierarchy specified therein.

2.9.2 File Context Managers

File context manager allows us to wrap file-processing code in a logic layer that ensures that the file will be closed automatically on exit, instead of relying on the auto-close during garbage collection.

```
with open('filename.txt') as f:
    for line in f:
        ...
```

2.9.3 Storing Native Python Objects: `pickle`

The `pickle` module is a more advanced tool that allows us to store almost any Python object in a file directly, with no to- or from-string conversion requirement on our part. Table 2.17 summarizes common methods. Note that files used to store the pickled object should be opened in binary mode, because the pickler creates and uses a `bytes` string object.

3 Statements and Syntax

3.1 Assignments Expressions and Prints

3.1.1 Assignment Statement Forms

Table 3.1 illustrates the different assignment statement forms in Python.

3.2 if Tests and Syntax Rules

3.2.1 The if/else Ternary Expression

Python runs expression `x` only `condition` turns out to be `True`, and runs expression `y` only if `condition` turns out to be `False`. That is, it short-circuits.

```
a = x if condition else y
```

3.3 while and for Loops

3.3.1 Loop else

The loop else block runs if and only if the loop is exited normally, i.e., without hitting a break. For example, the following piece of code determines whether a positive integer `y` is prime by searching for factors greater than 1:

```
x = y // 2  # For some y > 1
while x > 1:
    if y % x == 0:  # Remainder
        print(y, 'has factor', x)
```

Table 3.1
Assignment statement forms.

Form	Example
Basic	`spam = 'Spam'`
Tuple assignment	`spam, ham = 'yum', 'YUM'`
List assignment	`[spam, ham] = ['yum', 'YUM']`
Sequence assignment	`a, b, c, d = 'spam'`
Extended sequence unpacking	`a, *b = 'spam'`
Multiple-target assignment	`spam = ham = 'lunch'`
Augmented assignment	`spams += 42`

Table 3.2
Built-in functions for `for` loops.

Function	Result
`enumerate(sequence, start=0)`	The `next()` method of the iterator returned by `enumerate()` returns a count (start with `start`) and the value obtained from iterating over the `sequence`.
`range(start=0, stop, step=1)`	Return a list of `[start, start+step, start+2*step, ...]`. If `step` > 0, the last element is the largest `start+i*step` < `stop`; if `step` < 0, the last element is the smallest `start+i*step` > `stop`.
`zip([iterable,...])`	Return a list of tuples, where the `i`-th tuple contains the `i`-th element from each of the argument sequences or iterables.

```
        break  # Skip else
    x -= 1
else:  # Normal exit
    print(y, 'is prime')
```

3.3.2 Loop Coding Techniques

Python provides a set of built-ins that allow you to specialize the iteration in a `for` loop, as illustrated in Table 3.2.

3.4 Iterations and Comprehensions

3.4.1 Manual Iteration: `iter()` and `next()`

DEFINITION 3.1 **Iterable Object** An object is considered iterable if it is either a physically stored sequence, or an object that produces one result at a time in the context of an iteration tool like a `for` loop. Specifically, an iterable object supports the `iter()` call.

DEFINITION 3.2 **Iterator** An object returned by an iterable object on `iter()` that supports the `next()` call.

The following interaction demonstrates the equivalence between automatic and manual iteration.

```
# Automatic iteration.
for x in L:
    print(x)

# Manual iteration.
```

```
it = iter(L)
while True:
    try:
        x = next(iter)
        print(x)
    except StopIteration:
        break
```

3.5 The Documentation Interlude

4 Functions and Generators

4.1 Function Basics

4.1.1 Coding Functions

`def` is executable code. When Python reaches and runs a `def` statement, it creates an object and assigns it to a name.

4.2 Scopes

4.2.1 Python Scope Basics

Variables may be assigned in three different places, corresponding to three different scopes.

- If a variable is assigned inside a `def`, it is local to that function.
- If a variable is assigned in an enclosing `def`, it is nonlocal to nested functions.
- If a variable is assigned outside all `def`s, it is global to the entire file.

4.2.2 `global` and `nonlocal` Statements

`global` declares module-level variables that are to be assigned. By default, all names assigned in a function are local to that function and exist only while the function runs.

`nonlocal` declares enclosing function variables that are to be assigned. This allows enclosing functions to serve as a place to retain state, without using shared global names.

4.2.3 Factory Functions: Closures

DEFINITION 4.1 **Closure/Factory Function** A functional programming technique or a design pattern, describing the case where the function object in question remembers values in enclosing scopes regardless of whether those scopes are still present in memory.

Closure functions often provide a lighter-weight and viable alternative to classes when retaining state is the only goal.

```
1 def maker(N):
2     return lambda: x: x**N
3
4 f = maker(4)
5 print(f(4))
```

Table 4.1
Function argument-matching forms.

Syntax	Interpretation
Caller side	
`f(val)`	Normal argument: matched by position
`f(name=val)`	Keyword argument: matched by name
`f(*iterable)`	Pass all objects in `iterable` as individual positional arguments
`f(**dict)`	Pass all key/value pairs in `dict` as individual keyword arguments
Function side	
`def f(name)`	Normal argument: matches any passed value by position or name
`def f(name=val)`	Default argument value, if not passed in the call
`def f(*name)`	Matches and collects remaining positional arguments in a tuple
`def f(**name)`	Matches and collects remaining keyword arguments in a dictionary
`def f(*other, name)`	Arguments that must be passed by keyword only in calls
`def f(*, name=value)`	Arguments that must be passed by keyword only in calls

4.3 Arguments

4.3.1 Argument-Passing Basics

Arguments are passed by position, unless you say otherwise. Arguments are passed by assignment (object reference), i.e., the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

4.3.2 Special Argument-Matching Modes

Table 4.1 summarizes the syntax that invokes the special argument-matching modes.

Table 4.2
Functional programming tools.

Operation	Result
`map(function, iterable)`	Apply `function` to every element of `iterable` and return a list of the results.
`filter(function, iterable)`	Construct a list from elements of `iterable` for which `function` returns `True`. If `function` is `None`, all elements of `iterable` that are `False` are removed.
`reduce(function, iterable[, initializer])`	Apply `function` of two arguments cumulatively to the elements of `iterable`, from left to right, so as to reduce `iterable` to a single value. If `initializer` is given, it is placed before the items of `iterable` in the calculation, and serves as a default when `iterable` is empty.

4.4 Advanced Function Topics

4.4.1 Anonymous Functions: lambda

Defaults work on lambda arguments, just like in a `def`:

```
f = lambda a, b='fie', c='foe': a + b + c
```

4.4.2 Functional Programming Tools

Table 4.2 summarizes functional programming tools in Python.

4.5 Comprehensions and Generations

4.5.1 Comprehensions

The general structure of list comprehensions looks like the following. `map()` calls can be twice as fast as equivalent `for` loops, and list comprehensions are often faster than `map()` calls.

```
[expression for target1 in iterable1 if condition1
            for target2 in iterable2 if condition2]
```

4.5.2 Generator Functions and Expressions

Generator functions are coded as normal def statements, but use `yield` statements to return results one at a time, suspending and resuming their state between each.

Generator expressions are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list.

4.6 The Benchmarking Interlude

4.6.1 Timing Iteration Alternatives

```
import time
tic = time.perf_counter()
...
toc = time.perf_counter()
print(toc - tic)
```

```
import timeit
times = timeit.repeat(stmt='...', number=1000, repeat=5)
print(sum(times) / len(times))
```

4.6.2 Profiling

```
python -m cProfile main.py args
```

5 Modules and Packages

5.1 Modules The Big Picture

5.1.1 The Module Search Path

Python's module search path is in `sys.path`, which is a mutable list of directory name strings. Python first looks for the imported file in the home directory (directory containing your program's top-level script file). Next, Python searches all directories listed in your `PYTHONPATH` environment variable setting. Then, Python automatically searches the directories where the standard library modules are installed on your machine. Finally, Python searches the contents of any `.pth` files (if present).

5.2 Module Coding Basics

5.3 Module Packages

DEFINITION 5.1 **Package** A directory of Python code. A package import turns a directory on your computer into another Python namespace, with attributes corresponding to the subdirectories and module files that the directory contains.

5.3.1 Package Import Basics

For example, assume `dir1` resides within some container directory `dir0`, which is a component of the normal Python module search path. Package import has the form:

```
import dir1.dir2.mod
```

Package `__init__.py` Files. At least until Python 3.3, each directory named within the path of a package import statement must contain a file named `__init__.py`, or your package imports will fail. That is, in the example we have been using, both `dir1` and `dir2` must contain `__init__.py`, while `dir0` does not require such a file because it is not listed in the `import` statement itself. The `__init__.py` files serve primarily as a hook for performing initialization steps required by the package. These files can also be completely empty, though.

5.3.2 Python3.3 Namespace Packages

Python3.3 has four import models.

- **Basic module imports.** For example, `import mod, from mod import attr`. It imports files and their contents relative to the `sys.path` module search path.

- **Package imports.** For example, `import dir1.dir2.mod, from dir1.mod import attr`. It from give directory path extensions relative to the `sys.path` module search path, where each package is contained in a single directory and has an initialization file.
- **Package-relative imports.** For example, `from . import mod` (relative), `import mod` (absolute). It is used for intrapackage imports of the prior section, with its relative or absolute lookup schemes for dotted and nondotted imports. Import with dots will search for modules inside the package directory only, while imports without dots will skip the containing package itself and look elsewhere on the `sys.path` search path.
- **Namespace packages.** For example, `import splitdir.mod`. The new namespace package model which allows packages to span multiple directories, and requires no initialization file, introduced in Python3.3.

5.4 Advanced Module Topics

6 Classes and OOP

6.1 OOP The Big Picture

6.2 Class Coding Basics

6.3 A More Realistic Example

To extend inherited methods, we prefer simply calling the original through the superclass name `Superclass.method(...)` instead of `super()`.

6.4 Class Coding Details

6.4.1 The `class` Statement

Like a `def`, a `class` statement is an object builder, and an implicit assignment. When run, it generates a class object and stores a reference to it in the name used in the header. The general form is as follows.

```
class name(superclass, ...):
    attr = value  # Shared class data
    def method(self, ...):
        self.attr= value  # Per-instance data
```

6.4.2 Methods

A method's first argument always receives the instance object that is the implied subject of the method call. That is to say, method calls made through an instance are automatically translated to class method function calls of the following form:

```
instance.method(args, ...)
classname.method(instance, args, ...)
```

6.4.3 Inheritance

Abstract superclasses. Abstract superclasses, a.k.a. abstract base classes (ABC), require methods to be filled in by subclasses, are implemented with special class syntax. Coded this way, a class with an abstract method cannot be instantiated unless all of its abstract methods have been defined in subclasses.

```
import abc
class AbstractBaseClass(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def method(sef, ...):
        pass
```

6.5 Operator Overloading

6.5.1 The Basics

DEFINITION 6.1 **Operator Overloading** Intercept built-in operations in a class's methods.

Table 6.1 summarizes common operator overloading methods. The following is an example.

```
class A:
    def __getitem__(self, index):
        if isinstance(index, int):
            print('Index', index)
        else:
            print('Slicing', index.start, index.stop, index.step)

    def __iter__(self):
        return self
    def __next__(self):
        if ...:
            raise StopIteration
        else:
            return ...
```

6.6 Designing with Classes

6.6.1 Python and OOP

Python's implementation of OOP can be summarized by three ideas.

- **Inheritance.** Inheritance is based on attribute lookup in Python (in `x.name` expressions).
- **Polymorphism.** In `x.method`, the meaning of `method` depends on the type (class) of subject object `x`.
- **Encapsulation.** Methods and operators implement behavior, though data hiding is a convention by default.

Table 6.1
Common operator overloading methods.

Method	Implements	Called for
Constructors		
`__init__`	Constructor	Object creation `x = Class(args)`
`__del__`	Destructor	Object reclamation of `x`
Instance control		
`__call__`	Function calls	`x(*args, **kw)`
`__contains__`	Membership test	`item in x`
`__len__`	Length	`len(x)`
`__iter__`	Iteration contexts	`it = iter(x)`
`__next__`	Iteration contexts	`next(it)`
Arithmetic operations		
`__add__`	Operator +	`x + y, x += y`
`__bool__`	Boolean tests	`bool(x)`, truth tests
`__lt__`	Comparisons	`x < y`
`__or__`	Operator |	`x | y, x |= y`
`__repr__`	Conversions	`repr(x)`
`__str__`	Printing	`print(x), str(x)`
Attributes		
`__setattr__`	Attribute assignment	`x.any=value`
`__getattr__`	Attribute fetch	`x.undefined`
`__delattr__`	Attribute deletion	`del x.any`
Indexing and slicing		
`__setitem__`	Index and slice assignment	`X[k] = value, X[i:j] = iterable`
`__getitem__`	Indexing, slicing, iteration	`x[k], x[i, j]`
`__delitem__`	Indexing and slicing deletion	`del x[k], del x[i, j]`

Polymorphism means interfaces, not call signatures. Python does not support overloading functions based on the type signatures of their arguments (i.e., the number arguments passed and/or their type) like in C++. There can be only one definition of a particular method name. Instead, you should write your code to expect only an object interface, not a specific data type, which will be useful for a broader category of types and applications.

6.6.2 OOP and Inheritance: "Is-a" Relationships

6.6.3 OOP and Composition: "Has-a" Relationships

6.6.4 OOP and Delegation: "Wrapper" Proxy Objects

DEFINITION 6.2 **Delegation** Case where controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as logging or validating accesses, adding extra steps to interface components, or monitoring active instances.

Remark 6.1 Delegation is a special form of composition, with a single embedded object managed by a wrapper/proxy class that retains most or all of the embedded object's interface.

The following is an example, where `getattr(x, n)` is like `x.n`, except that `n` is an expression that evaluates to a string at runtime, not a variable.

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object  # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)  # Trace fetch
        return getattr(self.wrapped, attrname)  # Delegate fetch

x = Wrapper([1, 2, 3])  # Wrap a list
x.append(4)  # Delegate to list method
```

6.6.5 Pseudoprivate Class Attributes

In fact, attributes are all `public` and `virtual` in C++ terms; they are all accessible everywhere and are looked up dynamically at runtime.

6.6.6 Methods Are Objects: Bound or Unbound

There actually are two flavors in Python.

- **Unbound (class) method objects: no `self`.** An unbound method is the same as a simple function and can be called through the class's name.
- **Bound (instance) method objects: `self` + function pairs.**

6.7 Advanced Class Topics

7 Exceptions and Tools

7.1 Exception Basics

7.2 Exception Coding Details

7.2.1 The `try/except/else/finally` Statement

Here is the general and most complete format.

```
try:
    ...
except name1:
    ...
except (name2, name3):  # Run if any of these exceptions occur
    ...
except:  # Run for all other exceptions raised
    ...
else:  # Run if no exception was raised during try block
    ...
finally:  # Always run this code on the way out
    ...
```

7.2.2 The `raise` Statement

```
raise IndexError
```

7.2.3 The `assert` Statement

`assert` is used for debugging. Use a command line option like `python -O` to run in optimized mode and disable (and hence skip) asserts.

```
assert condition, error_msg
```

7.3 Exception Objects

7.4 Designing with Exceptions

II PYTHON STANDARD LIBRARIES

8 String Services

8.1 `string` Module

8.1.1 String constants

Table 8.1 summaries string constants.

8.1.2 Template Strings

`string.Template` provide simpler string substitutions. Instead of the normal %-based substitutions, `string.Template` support $-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `$identifier` names a substitution placeholder matching a mapping key of `identifier`.
- `${identifier}` is equivalent to `$identifier`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `'${noun}ification'`.

`string.Template(template)` takes the template string as arguments.

`string.Template.substitute(mapping[, **kw])` performs the template substitution, returning a new string. `mapping` is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where they keywords are the placeholders. When both `mapping` and `kw` are given and there are duplicates, the placeholders from `kw` take precedence.

`string.Template.safe_substitute(mapping[, **kw])` is like `string.Template.substitute()`, except that if placeholders are

Table 8.1
The constants defined in the `string` module.

Constant	Meaning
`string.ascii_lowercase`	`'abcdefghijklmnopqrstuvwxyz'`
`string.ascii_uppercase`	`'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
`string.ascii_letters`	`string.ascii_lowercase + string.ascii_uppercase`
`string.digits`	`'0123456789'`
`string.hexdigits`	`'0123456789abcdefABCDEF'`
`string.punctuation`	String of ASCII characters which are considered punctuation characters.
`string.printable`	String of characters which are considered printable.
`string.whitespace`	String containing all characters that are considered whitespace.

missing from `mapping` and `kw`, instead of raising a `KeyError` exception, the original placeholder will appear in the resulting string inact.

```
>>> import string
>>> template = string.Template('$who likes $what')
>>> template.substitute(who='tim', what='kung pao')
'tim likes kung pao'

>>> d = dict(who='tim')
>>> template.safe\_substitute(d)
'tim likes $what'
```

8.2 `re` Module

8.2.1 Regular Expression Syntax

DEFINITION 8.1 **Regular Expression** A regular expression specifies a set of strings that matches it.

Regular expressions use \ to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals. For example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be \\, and each backslash must be expressed as \\. The solution is the use Python's raw string notation for regular expression patterns `r'\\'`.

Regular expresssions can contain both special and ordinary characters. See Table 8.2 for sepcial characters.

Table 8.2

Regular expression special characters. Characters can be matched by complementing the set like `[^0-9]`.

Character	Meaning
`.`	Match any character except a newline.
`*`	Greedy match ≥ 0 repetitions of the receding RE.
`+`	Greedy match ≥ 1 repetitions of the receding RE.
`?`	Greedy match 0 or 1 repetitions of the receding RE.
`*?, +?, ??`	Non-greedy/Minimal match of `*`, `+`, and `?`.
`^`	Match the start of the string.
`$`	Match the end of the string.
`{m}`	Match exactly `m` copies of the preceding RE.
`{m, n}`	Greedy match [`m`, `n`] copies of the preceding RE. `m` defaults to be `0`; `n` defaults to be infinite upper bound.
`{m, n}?`	Non-greedy/Minimal match [`m`, `n`] copies of the preceding RE.
`\`	Either escapes special characters, or signals a sepcial sequence.
`[]`	Indicate a set of characters. Characters can be listed individually like `[amk]`. or use `-` to specify a range of characters like `[0-9A-Za-z]`.
`|`	Match either of the two operands.
`(...)`	Match the RE inside the parentheses, and indicates the start and end of a group.
`(?...)`	The first character after `?` determines what the meaning the construct is.
`\w`	`[a-zA-Z0-9_]`.
`\W`	`[^a-zA-Z0-9_]`.
`\s`	Any whitespace character `[\t\n\r\f\v]`.
`\S`	Any non-whitespace character `[^\t\n\r\f\v]`.

8.2.2 `re` Methods

`re.compile(pattern, flags=0)` compiles a RE pattern into a RE object, which can be used for `match()` and `search()` The following two snippets are equivalent:

```
>>> pattern = re.compile(pattern)
>>> result = pattern.match(string)

>>> result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting RE object for reuse is more efficient when the expression will be used several times in a single program.

`re.search(pattern, string, flags=0)` scans through `string` looking for the first location where the RE `pattern` produces a match, and return a corresponding `MatchObject` instance. Return `None` if no position in the string matches the pattern.

`re.split(pattern, string, maxsplit=0, flags=0)` split `string` by the occurrences of `pattern`. If capturing parentheses are used in `pattern`, then the text of all groups in the `pattern` are also returned as part of the resulting list. If `maxsplit` is nonzero, $\leq$ `maxsplit` splits occur, and the remainder of the string is returned as the final element of the list.

```
>>> re.split('\W+', 'words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'words, words, words.')
['Words', ', ', 'words', ', ', ''words', '.', '']
```

`re.escape(pattern)` escapes all the characters in `pattern` except ASCII letters and numbers. This is useful if you want to match an arbitrary literal string that may have RE metacharacters in it.

```
>>> re.escape('python.exe')
'python\.exe'
>>> legal_chars = string.ascii_lowercase + string.digits + "!#$
    %&'*+-.^_`|~:"
>>> re.escape(legal_chars)
'abcdefghijklmnopqrstuvwxyz0123456789\!\#\$\%\&\'\*\+\-\.\^\_\`\|\~\:'
```

9 Operating System Interfaces

`os` module provides a portable way of using operating system dependent functionality. All functions in this module raise `OSError` in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

See Table 9.1 for common methods.

Table 9.1
Common `os` methods.

Method	Result
Shell Commands	
`os.system(command)`	Execute the command in a subshell, and the return value is the exit status of the command.
`os.popen(command, mode='r')`	Open a pipe to or from `command`. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is `'r'` or `'w'`. The exit status of the command is available as the return value of the `close()` method of the file object, except that when the exit status is zero (termination without errors), `None` is returned.
Pathname Manipulations	

III SCIPY

10 NumPy

10.1 NumPy and `ndarray`

NumPy:

- `ndarray`: A fast and space-efficient multi-dimensional array providing vectorized arithmetic operations and sophisticated broadcasting capabilities.
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading/writing array data to disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- Tools for integrating code written in C, C++, and Fortan.

In NumPy dimensions are called axes. The number of axes is rank. The `ndarray` internally consists of the folowing:

- A pointer to data, which is a block of a system memory.
- The data type or `dtype`.
- A tuple indicating the array's shape.
- A tuple of strides, integers indicating the number of bytes to "step" in order to advance one element along a dimension.

strides are the critical ingredient in constructing copyless array views. Strides can be negative which enables an array to move backward through memory, which would be the case in a slice like `A[:, ::-1]`.

`dtype` is a special object describing the data type of the array. 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. The NumPy data types include: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float16`, `float32`, `float64`, `float128`, `complex64`, `complex128`, `complex256`, `bool`, `string_`. The Python's standard `float` type takes up 64 bits. The trailing underscores are used to avoid variable name conflicts between the NumPy specific types and the Python built-in ones.

10.2 NumPy Methods

See Table 10.1,10.2.10.3,10.4,10.5,10.6 for NumPy methods. The followings are some examples:

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
```

```
ValueError: arrays must have same number of dimensions

>>> a = np.array([[1, 1], [2, 2], [3, 3]], axis=None)
>>> np.insert(a, 1, 5)
array([1, 5, 1, 2, 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1], [2, 5, 2], [3, 5, 3]])
>>> np.insert(a, 1, [[1], [2], [3]], axis=1)
array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> np.delete(a, 1, axis=0)
array([[1, 2, 3, 4], [9, 10, 11, 12]])
>>> np.delete(a, [1, 3, 5], axis=None)
array([1, 3, 5, 7, 8, 9, 10, 11, 12])

>>> # Boolean Indexing
>>> # Suppose each name corresponds to a row in the data array.
>>> names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe
    '])
>>> data = np.random.randn(7, 4)
>>> data[names == 'Bob']  # A 2*4 array
>>> data[names == 'Bob', 2:]
>>> data[(names == 'Bob') | (names == 'Will')]
>>> data[data < 0] = 0
```

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. The fancy indexing, unlike slicing, always copies the data into a new array. To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order.

```
>>> A = np.random.randn(8, 4)
>>> A[[4, 3, 0, 6]]
>>> # Using negative indices select rows from the end.
>>> A[[-3, -5, -7]]
```

Passing multiple index arrays does something slightly different. It selects a 1-d array of elements corresponding to each tuple of indices.

```
>>> A[[1, 5, 7, 2], [0, 3, 1, 2]]
# A[1, 0], A[5, 3], A[7, 1], A[2, 2] are selected
```

Get a rectangular region formed by selecting a subset of the matrix's rows and columns.

```
>>> A[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

Besides, the sub-module `np.linalg` implements basic linear algebra, such as solving linear systems, SVD, etc. However, it is not guaranteed to be complied using efficient routines, and thus we recommend the use of `scipy.linalg`.

10.3 Performance Tips

1. Convert Python loops and conditional logic to array operations and boolean array operations.

2. Use broadcasting whenever possible.

3. Avoid copying data using array views (slicing). The result is not guaranteed to be contiguous.

4. Utilize ufuncs and ufunc methods.

5. Contiguous memory is important. Operations accessing contiguous blocks of memory (e.g., summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the ultrafast L1 or L2 cache.

Table 10.1

Numpy methods. If `dtype` is not given, the type will be determined as the minimum type required to hold the object. `order='C'` means to read/write the elements using C-like index order, with the last axis index changing fastest. `order='F'` means to read/write the elements using Fortran-like index order, with the first axis index changing fastest.

Method	Result
Inspecting Properties	
`np.ndarray.astype(dtype)`	Copy of the array (even `dtype` is the same as the old one), cast to a specified type.
`np.ndarray.dtype`	Return data type of the array's elements
`np.ndarray.ndim`	Number of array dimensions.
`np.ndarray.size`	Return number of elements in the array.
`np.ndarray.shape`	Return the shape of an array.
Creating Arrays	
`np.array(object, dtype=None, copy=True, order='C')`	`object` can be an array, or any (nested) sequence. This argument can only be used to upcast the array. For downcasting, use the `.astype()` method.
`np.zeros(shape, dtype=float, order='C')`	Return a new array of given shape and type, filled with zeros. `shape` is `int` or sequence of `int`s.
`np.zeros_like(a)`	Return an array of zeros with the same shape and type as a given array.
`np.ones(shape, dtype=float, order='C')`	Return a new array of given shape and type, filled with ones.
`np.ones_like(a)`	Return an array of ones with the same shape and type as a given array.
`np.full(shape, fill_value, dtype=None, order='C')`	Return a new array of given shape and type, filled with `fill_value`.
`np.eye(N, M=None, k=0, dtype=float)`	Return a 2-d array with ones on the `k`-th diagonal and zeros elsewhere. `N` specifies number of rows in the output. `M` specifies number of columns in the output. If `None`, defaults to `N`. `k` specifies the index of the diagonal. `0` refers to the main diagonal, `>0` refers to an upper diagonal, and `<0` refers to an lower diagonal.
`np.diag(a, k=0)`	If `a` is a 2-d array, return a copy of its `k`-th diagonal. If `a` is a 1-d array, return a 2-d array with `a` on the `k`-th diagonal. `k`> 0 for diagonals above the main diagonal, and `k`< 0 for diagonals below the main diagonal.
`np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`	Return evenly spaced numbers over a specified interval. `num` specifies number of samples to generate. The step size changes depends on `endpoint`. If `endpoint` is `False`, the sequence consists of all but the last of `num`+1 evenly spaced samples, so that `stop` is excluded. If `retstep` is true, return `(samples, step)`, where `step` is the spacing between samples.
`np.arange(start=0, stop, step=1, dtype=None)`	Return evenly spaced values within a given interval. When `np.arange()` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the `np.linspace()` function that receives as an argument the number of elements that we want, instead of the `step`.

Table 10.2

Numpy methods (continued). `size` is a tuple of `int`s sepcifing the output shape. The computatione is taken over the flattened array if `axis` is `None`, otherwise over the specified axis. `axis` can be `int` or tuple or `int`s. If `axis` is a tuple or `int`s, the computation is performed over multiple axes.

Method	Result
Random Numbers	
`np.random.rand(d0, d1, ..., dn)`	Return random values from uniform distribution [0, 1) in a given shape.
`np.random.randn(d0, d1, ..., dn)`	Return random values from standard normal distributions in a given shape.
`np.random.randint(low, high=None, size)`	Retun random integers from discrete uniform distribution [`low`, `high`). If `high` is `None`, then results are from [`0`, `low`).
`np.random.uniform(low=0.0, high=1.0, size=None)`	Draw samples from a uniform distribution [`low`, `high`).
`np.random.normal(loc=0.0, scale=1.0, size=None)`	Draw samples from a normal (Gaussian) distribution N(`loc`, `scale`^{2}).
`np.random.permutation(x)`	Ramdonly permute a sequence, or return a permuted range. If `x` is a multi-dimensional array, it is only shuffled along its first index.
`np.random.shuffle(x)`	Modify a sequence in place by shuffling its contents.
Math	
`np.abs(a)`	Return the absolute value of an array, element-wise.
`np.ceil(a)`	Return the ceiling of an array, element-wise.
`np.dot(a, b)`	Dot product of two arrays. For 1-d arrays it is the inner product of vectors.
`np.floor(a)`	Return the floor of an array, element-wise.
`np.log(a)`	Return the natural logarithm of an array, element-wise. Similar for `np.log2(a)` and `np.log10(a)` for 2-based and 10-based logarithms.
`np.log1p(a)`	Return the natural logarithm of one plus the array, element-wise.
`np.round(a, decimals=0)`	Round an array to the given number of decimals, element-wise.
`np.sin(a)`	Return the trigonometric sine of an array, element-wise.
`np.sqrt(a)`	Return the positive square-root of an array, element-wise.
`np.maximum(a, b)`	Return a new array containing the element-wise maximum of array elements.
`np.minimum(a, b)`	Return a new array containing the element-wise minimum of array elements.
Logical Tests	
`a1 == a2`	Return array of `bool`s, checking equality element-wise. Similar for `!=`, `<`, `<=`, `>`, `>=`.
`np.array_equal(a1, a2)`	Return `True` if two arrays have the same shape and elements.
`np.all(a, axis=None, keep-dims=False)`	Test whether all array elements along a specified axis evaluate to `True`.
`np.any(a, axis=None, keep-dims=False)`	Test whether any array elements along a specified axis evaluate to `True`.

Table 10.3

Numpy methods (continued). The computatione is taken over the flattened array if `axis` is `None`, otherwise over the specified axis. `axis` can be `int` or tuple or `int`s. If `axis` is a tuple or `int`s, the computation is performed over multiple axes. For integer inputs, `dtype` defaults to be `float64`; for floating point inputs, it is the same as the input `dtype`.

Method	Result
Statistics	
`np.mean(a, axis=None, dtype=None, keepdims=False)`	Compute the arithmetic mean along the specified axis.
`np.median(a, axis=None, keep-dims=False)`	Compute the median along the specified axis.
`np.cumsum(a, axis=None, dtype=None)`	Compute the cumulative sum of the elements along a given axis. The result has the same size as `a`.
`np.sum(a, axis=None, dtype=None, keepdims=False)`	Sum of array elements along the specified axis. If `dtype` is `None`, the `dtype` of `a` is used.
`np.max(a, axis=None, keep-dims=False)`	Compute the maximum elements along the specified axis.
`np.argmax(a, axis=None)`	Return the indices of the maximum values along the specified axis.
`np.min(a, axis=None, keep-dims=False)`	Compute the minimum elements along the specified axis.
`np.argmin(a, axis=None)`	Return the indices of the miinmum values along the specified axis.
`np.var(a, axis=None, dtype=None, ddof=0, keep-dims=False)`	Compute the variance along the specified axis. `ddof` specifies delta degrees of freedom. The divisor used in the calculation is $N-$`ddof`, where N is the number of elements.
`np.std(a, axis=None, dtype=None, ddof=0, keep-dims=False)`	Compute the standard deviation along the specified axis.
`np.sort(a, axis=-1)`	Return a sorted copy of an array. If `axis` is `None`, the array is flattened before sorting. The default is `-1`, which sorts along the last axis.
`np.ndarray.sort(axis=-1)`	Sort an array in place.
`np.argsort(a, axis=-1)`	Return the indices that would sort an array.
`np.historgram(a, bins=10, range=None)`	Compute the histogram over the flattened array. `range` specifies the lower and upper range of the bins. If not provided, it is `(a.min(), a.max())`.

Table 10.4
Numpy methods (continued).

Method	Result
Copying/Sorting/Reshaping	
`np.may_share_memory(a, b)`	Return `True` if memory-bounds of `a` and `b` overlap.
`np.copy(a)`	Return a copy of the given array.
`np.ndarray.sort(axis=-1)`	Sort an array in place. Default to sort along the last axis.
`np.sort(a, axis=-1)`	Return a sorted copy of an array.
`np.ndarray.flatten(order='C')`	Return a copy of the array collapsed into one dimension.
`np.ndarray.T`	Return the transpose of the given array.
`np.ndarray.transpose(*axes)`	Return a view of the array with dimensions permuted. `axes` is a tuple of `ints`, or `n ints` indicating how to permute the axes. `i` in the `j`-th place in the tuple means arrays `i`-th axis becomes the output's `j`-th axis.
`np.reshape(a, newshape, order='C')`	Gives a new shape to an array without changing its data. One shape dimension can be `-1`, which means that the value is inferred from the length of the array and remaining dimensions.
`np.resize(a, new_shape)`	Return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copied of `a`.
`np.ndarray.resize(new_shape)`	Return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with zeros.
`a[:, np.newaxis]`	Adding a dimension.
Adding/Removing Elements	
`np.append(a, values, axis=None)`	Append values to the end of a copy of `a`. `values` must have the same shape as `a`, excluding `axis`. If `axis` is not specified, both `a` and `values` are flattened before use.
`np.insert(a, obj, values, axis=None)`	Insert values along the given axis before the given indices of a copy of `a`. `obj` defines the index before which `values` is inserted so that `a[..., obj, ...] = values` is legal. If `axis` is `None` then `a` is flattened first.
`np.delete(a, obj, axis=None)`	Return a new array with sub-arrays along an axis deleted. `obj` defines the slice to indicate which sub-arrays to remove. If `axis` is `None` then `a` is flattened first.

Table 10.5

NumPy methods (continued). A slicing operation creates a view on the original array. Thus the original array is not copied in memoery. When modifying the view, the original array is modified as well. When slicing, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices. NumPy arrays can also be indexed with boolean or integer arrays (masks). This method is called fancy indexing. It creates copies not views. The boolean array must be of the same length as the axis it is indexing. When combining multiple boolean conditions, use boolean arithmetic operators like `&` and `|`. The Python keywords `and` and `or` do not work with boolean arrays.

Method	Result
Combining/Splitting	
`np.concatenate((a1, a2, ...), axis=0)`	Join a sequence of arrays along an existing axis. `a1, a2, ...` must have the same shape, except in the dimension corresponding to `axis`.
`np.vstack(tup)`	Take a sequence of arrays and stack them vertically (row wise). Arrays in `tup` must have the same shape along all but the first axis.
`np.hstack(tup)`	Take a sequence of arrays and stack them horizontally (column wise). Arrays in `tup` must have the same shape along all but the second axis.
`np.split(a, indices_or_sections, axis=0)`	Split an array into multiple sub-arrays. If `indices_or_sections` is an integer `N`, the array will be divided into `N` equal arrays along `axis`. If such a split is not possible, an error is raised. If `indices_or_sections` is a 1-d array of sorted integers, the entries indicate where along `axis` the array is split (e.g., `[i, j]` results in `a[:i], a[i:j], a[j:]`). If an index exceeds the dimension of the array along `axis`, an empty sub-array is returned correspondingly.
`np.array_split(a, indices_or_sections, axis=0)`	Split an array into multiple sub-arrays. The only difference between this function and `np.split()` is that `np.array_split()` allows `indices_or_sections` to be an integer that does *not* equally divide the axis.
Indexing/Slicing/Subsetting: Assuming `a` is an 2-d array.	
`a[i, j] = x`	Assign array element on index `i, j` the value `x`.
`a[i:j, k]`	Return the elements on rows `i:j` at column `k`.
`a[i:j]`	Return the rows `i:j`. In multi-dimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions.
`a < x`	Return an array with boolean values.
`a[a < x]`	Return array elements `<x`.
`a < x & a > y`	Return an array with boolean values.
`a[[2, 2, 4, 2], :]`	Indexing can be done with an array of integers, where the same index is repeated several times.
`~a`	Invert a boolean array.

Table 10.6
NumPy methods (continued).

Method	Result
Importing and Exporting	
`np.load(file)`	Load array from `.npy` file.
`np.save(file, arr)`	Save an array to a binary file in `.npy` format. If `file` is a string, a `.npy` extension will be appended to the filename if it does not already have one.
`np.loadtxt(fname, comments='#', delimiter=' ', skiprows=0[, usecols])`	Load data from a text file. Each row in the text file must have the same number of values. All the characters occurring on a line after `comments` are discarded. Skip the first `skiprows` lines. The optional `usecols` is a tuple specifing which columns to read, with column index starts with 0.
`np.genfromtxt(fname, comments='#', delimiter=' ', skip_header=0, skip_footer=0, missing_values=None[, usecols])`	Load data from a text file, with missing values handled as specified. `skip_header` specifies number of lines to skip at the beginning of the file. `skip_footer` specifies number of lines to skip at the end of the file. `missing_values` specifies the set of values to be used as default when the data are missing.
`np.savetxt(fname, X, fmt='%.18e', delimiter=' ', header='', footer='', comments='# ')`	Save an array to a text file. `header` specifies the string that will be written at the beginning of the file. `footer` specifies the string that will be written at the end of the file. `comments` will be prepended to the `header` and `footer` strings, to mark them as comments.

10.4 Broadcasting

NumPy can do operations on arrays of different size if NumPy can transform these arrays so that they all have the same size. This conversion is called broadcasting.

The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

11 Matplotlib

11.1 Matplotlib Methods

`pyplot` provides a procedural interface to the `matplotlib` object-oriented plotting library. It is modeled closely after MATLAB. A figure in matplotlib means the whole window in the user interface. Within this figure there can be sub-plots. So far we have used implicit figure and axes explicitly. See Table 11.1. For example, each of the following is legal:

```
# Plot x and y using default line style and color.
plt.plot(x, y)
# Plot x and y using blue circle markers.
plt.plot(x, y, 'bo')
# Plot y using x as index array 0..N-1.
plt.plot(y)
# Plot y using x as index array 0..N-1, with red plusses.
plt.plot(y, 'r+')
# label if for auto legends.
plt.plot(x, y, 'go-', label='line 1', linewidth=2)

# Set the locations of the xticks.
plt.xticks(np.arange(5))
# Set the locations and labels of the xticks.
plt.xticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))
# Rotate long labels.
plt.xticks(
    np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'), rotation
    =17)
```

Table 11.1
Plotting methods.

Method	Result
Creating and Plotting a Figure	
`plt.figure(num=None, figsize=None)`	Create a new figure. If `num` is not provided, the figure number for the newly created figure will be incremented, starting from 1. If `num` is provided and a figure with this id already exists, make it active and returns a reference to it. If this figure does not exists, create it and returns it. `figsize` is a tuple of integers specifing the width and height in inches.
`plt.subplot(nrows, ncols, plot_number)`	Return a subplot axes positioned by the given grid definition. `plot_number` starts at 1, and increments across rows first.
`plt.plot(*args, **kw)`	Plot lines. `args` is a variable length argument, allowing for multiple `x`, `y` pairs with an optional format string. `kw` can be used to set `label`, `linewidth`, etc.
Figure Settings	
`plt.xlim(xmin, xmax)`	Set the `x` limits of the current axes. Setting limits turns autoscaling off the `x`-axis.
`plt.xticks(*args, **kw)`	Set the `x` limits of the current tick locations and labels. `kw` can be used to rotate long labels.
`plt.xlabel(s)`	Set the `x` axis label of the current axis.
`plt.legend(loc='upper right')`	Place a legend on the axes.
`plt.title(s)`	Set a title of the current axes.
`plt.colorbar()`	Add a colorbar to a plot.
Show/Saving	
`plt.show(block=True)`	Display all figures and block until the figures have been closed.
`plt.savefig(fname)`	Save the current figure. The output format is deduced from the extension of `fname`.
Other Types of Plots	
`plt.bar(left, height, width=0.8)`	Make a bar plot. `left` is a sequence of scalars specifing the `x` coordinates of the left sides of the bars. `height` is a sequence of scalars specifing the heights of the bars. `width` specifies the width of the bars.
`plt.hist(x, bins=10, range=None)`	Plot a histogram. `range` specifies the lower and upper range of the bins. If not provided, it is `(x.min(), x.max())`.
`plt.imshow(X)`	Display an image on the axes. `X` may be a float array (with each value range 0.0 to 1.0), a uint8 array, or a PIL image.
`plt.scatter(x, y, s=20, marker='o')`	Make a scatter plot of `x` vs `y`, where `x` and `y` are sequence like objects of the same lengths. `s` is the size of points^2.