1.1 Python Overview

Building data structures and algorithms requires that we communicate detailed in-
structions to a computer. An excellent way to perform such communications isusing a high-level computer langu
language was originally developed by Guido van Rossum in the early 1990s, and
has since become a prominently used language in industry and education. The sec-ond major version of the lang
(more speci·cally, Python 3.1 or later). The latest version of the language is freely
available at www.python.org, along with documentation and tutorials.

In this chapter, we provide an overview of the Python programming language,
and we continue this discussion in the next chapter, focusing on object-orientedprinciples. We assume that reade
ence, although not necessarily using Python. This book does not provide a com-
plete description of the Python language (there are numerous language referencesfor that purpose), but it does i

1.1.1 The Python Interpreter

Python is formally an interpreted language. Commands are executed through a
piece of software known as the Python interpreter . The interpreter receives a com-
mand, evaluates that command, and reports the result of the command. While theinterpreter can be used interac
tionally stored in a ·le named with the .pysuf·x (e.g., demo.py).

On most operating systems, the Python interpreter can be started by typing
python from the command line. By default, the interpreter starts in interactive
mode with a clean workspace. Commands from a prede·ned script saved in a·le (e.g., demo.py) are executed by
an argument (e.g., python demo.py), or using an additional -i· a gi no r d e rt o
execute a script and then enter interactive mode (e.g., python -i demo.py).

Many integrated development environments (IDEs) provide richer software
development platforms for Python, including one named IDLE that is includedwith the standard Python distributio
step-by-step execution of a program while examining key variable values.

The set and frozenset Classes
Python·s setclass represents the mathematical notion of a set, namely a collection
of elements, without duplicates, and without an inherent order to those elements.
The major advantage of using a set, as opposed to a list,i st h a ti th a sah i g h l y
optimized method for checking whether a speci·c element is contained in the set.This is based on a data structur
topic of Chapter 10). However, there are two important restrictions due to the
algorithmic underpinnings. The ·rst is that the set does not maintain the elements
in any particular order. The second is that only instances of immutable types can be
added to a Python set. Therefore, objects such as integers, ·oating-point numbers,
and character strings are eligible to be elements of a set. It is possible to maintain a
set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The
frozenset class is an immutable form of the settype, so it is legal to have a set of
frozensets.
Python uses curly braces {and}as delimiters for a set, for example, as {17}
or{
red
,
green
,
blue
}. The exception to this rule is that {}does not
represent an empty set; for historical reasons, it represents an empty dictionary(see next paragraph). Instead, th
If an iterable parameter is sent to the constructor, then the set of distinct elements
is produced. For example, set(
hello
)produces {
h
,
e
,
l
,
o
}.
The dict Class
Python·s dict class represents a dictionary ,o rmapping , from a set of distinct keys
to associated values . For example, a dictionary might map from unique student ID
numbers, to larger student records (such as the student·s name, address, and course
grades). Python implements a dictusing an almost identical approach to that of a
set, but with storage of the associated values.
A dictionary literal also uses curly braces, and because dictionaries were intro-
duced in Python prior to sets, the literal form {}produces an empty dictionary.
A nonempty dictionary is expressed using a comma-separated series of key:valuepairs. For example, the diction
ga
:
Irish
,
de
:

14.3.2 DFS Implementation and Extensions

We begin by providing a Python implementation of the basic depth-·rst search algorithm, originally described with pseudo-code in Code Fragment 14.4. Our DFS function is presented in Code Fragment 14.5.

```
defDFS(g, u, discovered):
···Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
discovered is a dictionary mapping each vertex to the edge that was used to
discover it during the DFS. (u should be ·discovered· prior to the call.)
Newly discovered vertices will be added to the dictionary as a result.
···
foreing.incident
edges(u): # for every outgoing edge from u
v=e . o p p o s i t e ( u )
ifvnot in discovered: # v is an unvisited vertex
discovered[v] = e # e is the tree edge that discovered v
DFS(g, v, discovered) # recursively explore from v
```

Code Fragment 14.5: Recursive implementation of depth-·rst search on a graph, starting at a designated vertex u.

In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, our implementation introduces a third parameter, nameddiscovered . This parameter sl graph to the tree edge that was used to discover that vertex. As a technicality, weassume that the source vertex value. Thus, a caller might start the traversal as follows:

```
result = {u:None} # a new dictionary, with u trivially discovered
DFS(g, u, result)
```

The dictionary serves two purposes. Internally, the dictionary provides a mecha-nism for recognizing visited verti values within the dictionary are the DFS tree edges at the conclusion of the process. Because the dictionary is hash-based, the test, · ifvnot in discovered ,· and the record-keeping step, · discovered[v] = e ,· run in O(1)expected time, rather than worst-case time. In practice, this is a compromise we are willing to accept,but it does violate the formal anal could be used as indices into an array-based lookup table rather than a hash-basedmap. Alternatively, we could tree edge directly as part of the vertex instance.

14.2.5 Python Implementation

In this section, we provide an implementation of the Graph ADT. Our implementa-
tion will support directed or undirected graphs, but for ease of explanation, we ·rstdescribe it in the context of an
We use a variant of the adjacency map representation. For each vertex v,w e
use a Python dictionary to represent the secondary incidence map I(v).H o w e v e r ,
we do not explicitly maintain lists Vand E, as originally described in the edge list
representation. The list Vis replaced by a top-level dictionary Dthat maps each
vertex vto its incidence map I(v); note that we can iterate through all vertices by
generating the set of keys for dictionary D. By using such a dictionary Dto map
vertices to the secondary incidence maps, we need not maintain references to thoseincidence maps as part of th
inO(1)expected time. This greatly simpli·es our implementation. However, a
consequence of our design is that some of the worst-case running time bounds forthe graph ADT operations, giv
than maintain list E, we are content with taking the union of the edges found in the
various incidence maps; technically, this runs in O(n+m)time rather than strictly
O(m)time, as the dictionary Dhas nkeys, even if some incidence maps are empty.
Our implementation of the graph ADT is given in Code Fragments 14.1 through
14.3. Classes Vertex andEdge , given in Code Fragment 14.1, are rather simple,
and can be nested within the more complex Graph class. Note that we de·ne the
hash
method for both Vertex andEdge so that those instances can be used as
keys in Python·s hash-based sets and dictionaries. The rest of the Graph class is
given in Code Fragments 14.2 and 14.3. Graphs are undirected by default, but canbe declared as directed with a
Internally, we manage the directed case by having two different top-level dictio-
nary instances,
outgoing and
incoming , such that
outgoing[v] maps to another
dictionary representing Iout(v),a n d
incoming[v] maps to a representation of Iin(v).
In order to unify our treatment of directed and undirected graphs, we continue touse the
outgoing and
incoming identi·ers in the undirected case, yet as aliases
to the same dictionary. For convenience, we de·ne a utility named is
directed to
allow us to distinguish between the two cases.
For methods degree andincident
edges , which each accept an optional param-
eter to differentiate between the outgoing and incoming orientations, we choose theappropriate map before proc
vertex , we always initial-
ize
outgoing[v] to an empty dictionary for new vertex v. In the directed case, we
independently initialize
incoming[v] as well. For the undirected case, that step is
unnecessary as
outgoing and
incoming are aliases. We leave the implementations
of methods remove
vertex andremove
edge as exercises (C-14.37 and C-14.38).

We see lazy evaluation used in many of Python·s libraries. For example, the dictionary class supports methods keys() ,values() ,a n ditems() , which respec- tively produce a ·view· of all keys, values, or (key,value) pairs within a dictionary. None of these methods produces an explicit list of results. Instead, the views thatare produced are iterable objec calling the listclass constructor with the iteration as a parameter. For example, the syntax list(range(1000)) produces a list instance with values from 0 to 999, while the syntax list(d.values()) produces a list that has elements based upon the current values of dictionary d. We can similarly construct a tuple orsetinstance based upon a given iterable.

Generators

In Section 2.3.4, we will explain how to de·ne a class whose instances serve as iterators. However, the most convenient technique for creating iterators in Python is through the use of generators . A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series. As an example, consider the goal of determining all factors of a positive integer. For example, the number 100 hasfactors 1, 2, 4, 5, 10, 20, 25, 50, return a list containing all factors, implemented as:

```
deffactors(n): # traditional function that computes factors
results = [ ] # store factors in a new list
forkinrange(1,n+1):
ifn%k= =0 : # divides evenly, thus k is a factor
results.append(k) # add k to the list of factors
return results # return the entire list
```

In contrast, an implementation of a generator for computing those factors could be implemented as follows:

```
deffactors(n): # generator that computes factors
forkinrange(1,n+1):
ifn%k= =0 : # divides evenly, thus k is a factor
yieldk # yield this factor as next result
```

Notice use of the keyword yield rather than return to indicate a result. This indi- cates to Python that we are de·ning a generator, rather than a traditional function. It is illegal to combine yield andreturn statements in the same implementation, other than a zero-argument return statement to cause a generator to end its execution. If a programmer writes a loop such as forfactorinfactors(100): , an instance of our generator is created. For each iteration of the loop, Python executes our procedure

Python carefully extends the semantics of //and%to cases where one or both
operands are negative. For the sake of notation, let us assume that variables n
andmrepresent respectively the dividend anddivisor of a quotientn
m,a n dt h a t
q=n/ /m andr=n%m . Python guarantees that q
m+r will equal n.W e
already saw an example of this identity with positive operands, as 6 ·4+3=27.
When the divisor mis positive, Python further guarantees that 0 ·r<m.A s
a consequence, we ·nd that ·27 // 4 evaluates to ·7and·27 % 4 evaluates
to1,a s (·7)·4+1=·27. When the divisor is negative, Python guarantees that
m<r·0. As an example, 27 // ·4is·7and27 % ·4is·1, satisfying the
identity 27 =(·7)·(·4)+(·1).
The conventions for the //and%operators are even extended to ·oating-
point operands, with the expression q=n/ /m being the integral ·oor of the
quotient, and r=n%m being the ·remainder· to ensure that q
m+r equals
n. For example, 8.2 // 3.14 evaluates to 2.0and8.2 % 3.14 evaluates to 1.92,a s
2.0·3.14+1.92=8.2.
Bitwise Operators
Python provides the following bitwise operators for integers:
· bitwise complement (pre·x unary operator)
& bitwise and
| bitwise or
· bitwise exclusive-or
<< shift bits left, ·lling in with zeros
>> shift bits right, ·lling in with sign bit
Sequence Operators
Each of Python·s built-in sequence types (str,tuple ,a n dlist) support the following
operator syntaxes:
s[j] element at index j
s[start:stop] slice including indices [start,stop)
s[start:stop:step] slice including indices start ,start + step ,
start + 2
step,...,u pt ob u tn o te q u a lling or stop
s+t concatenation of sequences
k
s shorthand for s+s+s+. . . (k times)
valins containment check
valnot in s non-containment check
Python relies on zero-indexing of sequences, thus a sequence of length nhas ele-
ments indexed from 0 to n·1 inclusive. Python also supports the use of negative
indices , which denote a distance from the end of the sequence; index ·1 denotes
the last element, index ·2 the second to last, and so on. Python uses a slicing

15.1.3 Additional Memory Used by the Python Interpreter
We have discussed, in Section 15.1.1, how the Python interpreter allocates memory
for objects within a memory heap. However, this is not the only memory that isused when executing a Python pro
The Run-Time Call Stack
Stacks have a most important application to the run-time environment of Python
programs. A running Python program has a private stack, known as the call stack
orPython interpreter stack , that is used to keep track of the nested sequence of
currently active (that is, nonterminated) invocations of functions. Each entry of
the stack is a structure known as an activation record orframe , storing important
information about an invocation of a function.
At the top of the call stack is the activation record of the running call ,t h a ti s ,
the function activation that currently has control of the execution. The remainingelements of the stack are activat
tions that have invoked another function and are currently waiting for that other
function to return control when it terminates. The order of the elements in the stack
corresponds to the chain of invocations of the currently active functions. When anew function is called, an activa
interpreter resumes the processing of the previously suspended call.
Each activation record includes a dictionary representing the local namespace
for the function call. (See Sections 1.10 and 2.5 for further discussion of name-
spaces). The namespace maps identi·ers, which serve as parameters and local
variables, to object values, although the objects being referenced still reside in the
memory heap. The activation record for a function call also includes a reference tothe function de·nition itself, an
to maintain the address of the statement within the function that is currently exe-
cuting. When one function returns control to another, the stored program counter
for the suspended function allows the interpreter to properly continue execution ofthat function.
Implementing Recursion
One of the bene·ts of using a stack to implement the nesting of function calls isthat it allows programs to use rec
self, as discussed in Chapter 4. We implicitly described the concept of the call
stack and the use of activation records within our portrayal of recursion traces in

1.1. Python Overview 3
1.1.2 Preview of a Python Program
As a simple introduction, Code Fragment 1.1 presents a Python program that com-
putes the grade-point average (GPA) for a student based on letter grades that areentered by a user. Many of the
Python·s syntax relies heavily on the use of whitespace. Individual statements
are typically concluded with a newline character, although a command can extendto another line, either with a co
delimiter has not yet been closed, such as the {character in de·ning value
map.
Whitespace is also key in delimiting the bodies of control structures in Python.
Speci·cally, a block of code is indented to designate it as the body of a controlstructure, and nested control struc
including a nested conditional structure.
Comments are annotations provided for human readers, yet ignored by the
Python interpreter. The primary syntax for comments in Python is based on useof the #character, which designa
print(
Welcome to the GPA calculator.
)
print(
 Please enter all your letter grades, one per line.
)
print(
 Enter a blank line to designate the end.
)
# map from letter grade to point value
points = {
A+
:4.0,
 A
:4.0,
 A-
:3.67,
 B+
:3.33,
 B
:3.0,
 B-
:2.67,
C+
:2.33,
 C
:2.0,
 C
:1.67,
 D+
:1.33,
 D
:1.0,
 F
:0.0}
num
courses = 0

1.10 Scopes and Namespaces

When computing a sum with the syntax x+y in Python, the names xandymust have been previously associated with objects that serve as values; a NameError will be raised if no such de·nitions are found. The process of determining the value associated with an identi·er is known as name resolution .

Whenever an identi·er is assigned to a value, that de·nition is made with a speci·c scope . Top-level assignments are typically made in what is known as global scope. Assignments made within the body of a function typically have scope that is local to that function call. Therefore, an assignment, x=5 , within a function has no effect on the identi·er, x, in the broader scope.

Each distinct scope in Python is represented using an abstraction known as a namespace . A namespace manages all identi·ers that are currently de·ned in a given scope. Figure 1.8 portrays two namespaces, one being that of a caller to our count function from Section 1.5, and the other being the local namespace during the execution of that function.

A-
str
A
str
CS
·oat
3.56int
2
itemdatagrades
majorgpatargetn
list
str
B+
str
A-
str

Figure 1.8: A portrayal of the two namespaces associated with a user·s call count(grades,
 A
), as de·ned in Section 1.5. The left namespace is the caller·s and the right namespace represents the local scope of the function.

Python implements a namespace with its own dictionary that maps each identifying string (e.g.,
 n
) to its associated value. Python provides several ways to examine a given namespace. The function, dir, reports the names of the identi·ers in a given namespace (i.e., the keys of the dictionary), while the function, vars, returns the full dictionary. By default, calls to dir() andvars() report on the most locally enclosing namespace in which they are executed.

1.3 Expressions, Operators, and Precedence

In the previous section, we demonstrated how names can be used to identify existing objects, and how literals and constructors can be used to create instances of built-in classes. Existing values can be combined into larger syntactic expressions using a variety of special symbols and keywords known as operators . The semantics of an operator depends upon the type of its operands. For example, when a andbare numbers, the syntax a+b indicates addition, while if aandbare strings, the operator indicates concatenation. In this section, we describe Python·s opera-tors in various contexts of the b
We continue, in Section 1.3.1, by discussing compound expressions ,s u c ha s
a+b
c, which rely on the evaluation of two or more operations. The order
in which the operations of a compound expression are evaluated can affect theoverall value of the expression. F
precedence for evaluating operators, and it allows a programmer to override this order by using explicit parentheses to group subexpressions.

Logical Operators

Python supports the following keyword operators for Boolean values:

not unary negation
and conditional and
or conditional or

Theand andoroperators short-circuit , in that they do not evaluate the second operand if the result can be determined based on the value of the ·rst operand. This feature is useful when constructing Boolean expressions in which we ·rst test that a certain condition holds (such as a reference not being None), and then test a condition that could have otherwise generated an error condition had the prior testnot succeeded.

Equality Operators

Python supports the following operators to test two notions of equality:

is same identity
is not different identity
== equivalent
!= not equivalent

The expression aisbevaluates to True , precisely when identi·ers aandbare aliases for the same object. The expression a= =b tests a more general notion of equivalence. If identi·ers aandbrefer to the same object, then a= =b should also evaluate to True .Y e ta= =b also evaluates to True when the identi·ers refer to