A\_Primer\_Scientific\_Programming\_Python\_5E\_Hans\_c03

ctions and Branching

This chapter introduces two fundamental and extremely useful concepts in pro-

gramming: user-defined functions and branching of program flow, the latter often

referred to as if tests. The programs associated with the chapter are found in the

folder src/funcif1.

3.1 Functions

In a computer language like Python, the term function means more than just a math-

ematical function. A function is a collection of statements that you can execute

wherever and whenever you want in the program. You may send variables to the

function to influence what is getting computed by statements in the function, and

the function may return new objects back to you.

In particular, functions help avoid duplicating code snippets by putting all similar

snippets in a common place. This strategy saves typing and makes it easier to

change the program later. Functions are also often used to just split a long program

into smaller, more manageable pieces, so the program and your own thinking about

it become clearer. Python comes with lots of pre-defined functions (math.sqrt,

range, and len are examples we have met so far). This section explains how you

can define your own functions.

3.1.1 Mathematical Functions as Python Functions

Let us start with making a Python function that evaluates a mathematical function,

more precisely the function F .C / for converting Celsius degrees C to the corre-

sponding Fahrenheit degrees F :

F .C / D

1

9

C C 32 :

5

http://tinyurl.com/pwyasaa/funcif

© Springer-Verlag Berlin Heidelberg 2016

H.P. Langtangen, A Primer on Scientific Programming with Python,

Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3\_3

9192

3

Functions and Branching

The corresponding Python function must take C as argument and return the value

F .C /. The code for this looks like

def F(C):

return (9.0/5)\*C + 32

All Python functions begin with def, followed by the function name, and then

inside parentheses a comma-separated list of function arguments. Here we have

only one argument C. This argument acts as a standard variable inside the function.

The statements to be performed inside the function must be indented. At the end of

a function it is common to return a value, that is, send a value “out of the function”.

This value is normally associated with the name of the function, as in the present

case where the returned value is the result of the mathematical function F .C /.

The def line with the function name and arguments is often referred to as the

function header, while the indented statements constitute the function body.

To use a function, we must call (or invoke) it. Because the function returns

a value, we need to store this value in a variable or make use of it in other ways.

Here are some calls to F:

temp1 = F(15.5)

a = 10

temp2 = F(a)

print F(a+1)

sum\_temp = F(10) + F(20)

The returned object from F(C) is in our case a float object. The call F(C) can

therefore be placed anywhere in a code where a float object would be valid. The

print statement above is one example.

As another example, say we have a list Cdegrees of Celsius degrees and we

want to compute a list of the corresponding Fahrenheit degrees using the F function

above in a list comprehension:

Fdegrees = [F(C) for C in Cdegrees]

Yet another example may involve a slight variation of our F(C) function, where

a formatted string instead of a real number is returned:

>>> def F2(C):

...

F\_value = (9.0/5)\*C + 32

...

return ’%.1f degrees Celsius corresponds to ’\

...

’%.1f degrees Fahrenheit’ % (C, F\_value)

...

>>> s1 = F2(21)

>>> s1

’21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit’

The assignment to F\_value demonstrates that we can create variables inside a func-

tion as needed.3.1

Functions

93

3.1.2 Understanding the Program Flow

A programmer must have a deep understanding of the sequence of statements that

are executed in the program and be able to simulate by hand what happens with

a program in the computer. To help build this understanding, a debugger (see

Sect. F.1) or the Online Python Tutor2 are excellent tools. A debugger can be used

for all sorts of programs, large and small, while the Online Python Tutor is primarily

an educational tool for small programs. We shall demonstrate it here.

Below is a program c2f.py having a function and a for loop, with the purpose

of printing out a table for conversion of Celsius to Fahrenheit degrees:

def F(C):

F = 9./5\*C + 32

return F

dC = 10

C = -30

while C <= 50:

print ’%5.1f %5.1f’ % (C, F(C))

C += dC

We shall now ask the Online Python Tutor to visually explain how the program

is executed. Go to http://www.pythontutor.com/visualize.html, erase the code there

and write or paste the c2f.py file into the editor area. Click Visualize Execution.

Press the forward button to advance one statement at a time and observe the evolu-

tion of variables to the right in the window. This demo illustrates how the program

jumps around in the loop and up to the F(C) function and back again. Figure 3.1

Fig. 3.1 Screen shot of the Online Python Tutor and stepwise execution of the c2f.py program

2

http://www.pythontutor.com/94

3

Functions and Branching

gives a snapshot of the status of variables, terminal output, and what the current and

next statements are.

Tip: How does a program actually work?

Every time you are a bit uncertain about the flow of statements in a program with

loops and/or functions, go to http://www.pythontutor.com/visualize.html, paste

in your program and see exactly what happens.

3.1.3 Local and Global Variables

Local variables are invisible outside functions Let us reconsider the F2(C) func-

tion from Sect. 3.1.1. The variable F\_value is a local variable in the function, and

a local variable does not exist outside the function, i.e., in the main program. We

can easily demonstrate this fact by continuing the previous interactive session:

>>> c1 = 37.5

>>> s2 = F2(c1)

>>> F\_value

...

NameError: name ’F\_value’ is not defined

This error message demonstrates that the surrounding program outside the function

is not aware of F\_value. Also the argument to the function, C, is a local variable

that we cannot access outside the function:

>>> C

...

NameError: name ’C’ is not defined

On the contrary, the variables defined outside of the function, like s1, s2, and c1

in the above session, are global variables. These can be accessed everywhere in

a program, also inside the F2 function.

Local variables hide global variables Local variables are created inside a func-

tion and destroyed when we leave the function. To learn more about this fact, we

may study the following session where we write out F\_value, C, and some global

variable r inside the function:

>>> def F3(C):

...

F\_value = (9.0/5)\*C + 32

...

print ’Inside F3: C=%s F\_value=%s r=%s’ % (C, F\_value, r)

...

return ’%.1f degrees Celsius corresponds to ’\

...

’%.1f degrees Fahrenheit’ % (C, F\_value)

...

>>> C = 60

# make a global variable C

>>> r = 21

# another global variable

>>> s3 = F3(r)3.1

Functions

95

Inside F3: C=21 F\_value=69.8 r=21

>>> s3

’21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit’

>>> C

60

This example illustrates that there are two C variables, one global, defined in the

main program with the value 60 (an int object), and one local, living when the

program flow is inside the F3 function. The value of this latter C is given in the call

to the F3 function (an int object). Inside the F3 function the local C hides the global

C variable in the sense that when we refer to C we access the local variable. (The

global C can technically be accessed as globals()[’C’], but one should avoid

working with local and global variables with the same names at the same time!)

The Online Python Tutor gives a complete overview of what the local and global

variables are at any point of time. For instance, in the example from Sect. 3.1.2,

Fig. 3.1 shows the content of the three global variables F, dC, and C, along with the

content of the variables that are in play in this call of the F(C) function: C and F.

How Python looks up variables

The more general rule, when you have several variables with the same name, is

that Python first tries to look up the variable name among the local variables,

then there is a search among global variables, and finally among built-in Python

functions.

Example Here is a complete sample program that aims to illustrate the rule above:

print sum

sum = 500

print sum

# sum is a built-in Python function

# rebind the name sum to an int

# sum is a global variable

def myfunc(n):

sum = n + 1

print sum # sum is a local variable

return sum

sum = myfunc(2) + 1

print sum

# new value in global variable sum

In the first line, there are no local variables, so Python searches for a global value

with name sum, but cannot find any, so the search proceeds with the built-in func-

tions, and among them Python finds a function with name sum. The printout of sum

becomes something like <built-in function sum>.

The second line rebinds the global name sum to an int object. When trying to

access sum in the next print statement, Python searches among the global variables

(no local variables so far) and finds one. The printout becomes 500. The call

myfunc(2) invokes a function where sum is a local variable. Doing a print sum

in this function makes Python first search among the local variables, and since sum96

3

Functions and Branching

is found there, the printout becomes 3 (and not 500, the value of the global variable

sum). The value of the local variable sum is returned, added to 1, to form an int

object with value 4. This int object is then bound to the global variable sum. The

final print sum leads to a search among global variables, and we find one with

value 4.

Changing global variables inside functions The values of global variables can

be accessed inside functions, but the values cannot be changed unless the variable

is declared as global:

a = 20; b = -2.5# global variables

def f1(x):

a = 21

return a\*x + b# this is a new local variable

print a# yields 20

def f2(x):

global a

a = 21

return a\*x + b

f1(3); print a

f2(3); print a

# the global a is changed

# 20 is printed

# 21 is printed

Note that in the f1 function, a = 21 creates a local variable a. As a programmer

you may think you change the global a, but it does not happen! You are strongly

encouraged to run the programs in this section in the Online Python Tutor, which

is an excellent tool to explore local versus global variables and thereby get a good

understanding of these concepts.

3.1.4 Multiple Arguments

The previous F(C) and F2(C) functions from Sect. 3.1.1 are functions of one vari-

able, C, or as we phrase it in computer science: the functions take one argument

(C). Functions can have as many arguments as desired; just separate the argument

names by commas.

Consider the mathematical function

1

y.t/ D v0 t gt 2 ;

2

where g is a fixed constant and v0 is a physical parameter that can vary. Mathemat-

ically, y is a function of one variable, t, but the function values also depends on the

value of v0 . That is, to evaluate y, we need values for t and v0 . A natural Python

implementation is therefore a function with two arguments:3.1

Functions

97

def yfunc(t, v0):

g = 9.81

return v0\*t - 0.5\*g\*t\*\*2

Note that the arguments t and v0 are local variables in this function. Examples on

valid calls are

y = yfunc(0.1, 6)

y = yfunc(0.1, v0=6)

y = yfunc(t=0.1, v0=6)

y = yfunc(v0=6, t=0.1)

The possibility to write argument=value in the call makes it easier to read and

understand the call statement. With the argument=value syntax for all arguments,

the sequence of the arguments does not matter in the call, which here means that

we may put v0 before t. When omitting the argument= part, the sequence of ar-

guments in the call must perfectly match the sequence of arguments in the function

definition. The argument=value arguments must appear after all the arguments

where only value is provided (e.g., yfunc(t=0.1, 6) is illegal).

Whether we write yfunc(0.1, 6) or yfunc(v0=6, t=0.1), the arguments

are initialized as local variables in the function in the same way as when we assign

values to variables:

t = 0.1

v0 = 6

These statements are not visible in the code, but a call to a function automatically

initializes the arguments in this way.

3.1.5 Function Argument or Global Variable?

Since y mathematically is considered a function of one variable, t, some may argue

that the Python version of the function, yfunc, should be a function of t only. This

is easy to reflect in Python:

def yfunc(t):

g = 9.81

return v0\*t - 0.5\*g\*t\*\*2

The main difference is that v0 now becomes a global variable, which needs to be

initialized outside the function yfunc (in the main program) before we attempt to

call yfunc. The next session demonstrates what happens if we fail to initialize such

a global variable:98

3

Functions and Branching

>>> def yfunc(t):

...

g = 9.81

...

return v0\*t - 0.5\*g\*t\*\*2

...

>>> yfunc(0.6)

...

NameError: global name ’v0’ is not defined

The remedy is to define v0 as a global variable prior to calling yfunc:

>>> v0 = 5

>>> yfunc(0.6)

1.2342

The rationale for having yfunc as a function of t only becomes evident in

Sect. 3.1.12.

3.1.6 Beyond Mathematical Functions

So far our Python functions have typically computed some mathematical function,

but the usefulness of Python functions goes far beyond mathematical functions.

Any set of statements that we want to repeatedly execute under potentially slightly

different circumstances is a candidate for a Python function. Say we want to make

a list of numbers starting from some value and stopping at another value, with

increments of a given size. With corresponding variables start=2, stop=8, and

inc=2, we should produce the numbers 2, 4, 6, and 8. Let us write a function

doing the task, together with a couple of statements that demonstrate how we call

the function:

def makelist(start, stop, inc):

value = start

result = []

while value <= stop:

result.append(value)

value = value + inc

return result

mylist = makelist(0, 100, 0.2)

print mylist # will print 0, 0.2, 0.4, 0.6, ... 99.8, 100

Remark 1 The makelist function has three arguments: start, stop, and inc,

which become local variables in the function. Also value and result are local

variables. In the surrounding program we define only one variable, mylist, and

this is then a global variable.

Remark 2 You may think that range(start, stop, inc) makes the makelist

function redundant, but range can only generate integers, while makelist can

generate real numbers too, and more, as demonstrated in Exercise 3.44.3.1

Functions

99

3.1.7 Multiple Return Values

Python functions may return more than one value. Suppose we are interested in

evaluating both y.t/ and y 0 .t/:

1

y.t/ D v0 t gt 2 ;

2

y 0 .t/ D v0 gt :

To return both y and y 0 we simply separate their corresponding variables by

a comma in the return statement:

def yfunc(t, v0):

g = 9.81

y = v0\*t - 0.5\*g\*t\*\*2

dydt = v0 - g\*t

return y, dydt

Calling this latter yfunc function makes a need for two values on the left-hand side

of the assignment operator because the function returns two values:

position, velocity = yfunc(0.6, 3)

Here is an application of the yfunc function for producing a nicely formatted

table of t, y.t/, and y 0 .t/ values:

t\_values = [0.05\*i for i in range(10)]

for t in t\_values:

position, velocity = yfunc(t, v0=5)

print ’t=%-10g position=%-10g velocity=%-10g’ % \

(t, position, velocity)

The format %-10g prints a real number as compactly as possible (decimal or sci-

entific notation) in a field of width 10 characters. The minus sign (-) after the

percentage sign implies that the number is left-adjusted in this field, a feature that

is important for creating nice-looking columns in the output:

t=0

t=0.05

t=0.1

t=0.15

t=0.2

t=0.25

t=0.3

t=0.35

t=0.4

t=0.45

position=0

position=0.237737

position=0.45095

position=0.639638

position=0.8038

position=0.943437

position=1.05855

position=1.14914

position=1.2152

position=1.25674

velocity=5

velocity=4.5095

velocity=4.019

velocity=3.5285

velocity=3.038

velocity=2.5475

velocity=2.057

velocity=1.5665

velocity=1.076

velocity=0.5855

When a function returns multiple values, separated by a comma in the return

statement, a tuple (Sect. 2.5) is actually returned. We can demonstrate that fact by

the following session:100

3

Functions and Branching

>>> def f(x):

...

return x, x\*\*2, x\*\*4

...

>>> s = f(2)

>>> s

(2, 4, 16)

>>> type(s)

<type ’tuple’>

>>> x, x2, x4 = f(2)

# store in separate variables

Note that storing multiple return values in separate variables, as we do in the last

line, is actually the same functionality as we use for storing list (or tuple) elements

in separate variables, see Sect. 2.2.1.

3.1.8 Computing Sums

Our next example concerns a Python function for calculating the sum

L.xI n/ D

n

X

1

i D1

i

x

1Cx

i

:

(3.1)

To compute a sum in a program, we use a loop and add terms to an accumulation

variable inside the loop. Section 2.1.4 explains the idea. However, summation

expressions with an integer counter, such as i in (3.1), are normally implemented

counter and not a while loop as in Sect. 2.1.4. For example,

by a for loop over the iP

the implementation of niD1 i 2 is typically implemented as

s = 0

for i in range(1, n+1):

s += i\*\*2

For the specific sum (3.1) we just replace i\*\*2 by the right term inside the for

loop:

s = 0

for i in range(1, n+1):

s += (1.0/i)\*(x/(1.0+x))\*\*i

Observe the factors 1.0 used to avoid integer division, since i is int and x may

also be int.

It is natural to embed the computation of the sum in a function that takes x and

n as arguments and returns the sum:

def L(x, n):

s = 0

for i in range(1, n+1):

s += (1.0/i)\*(x/(1.0+x))\*\*i

return s3.1

Functions

101

Our formula (3.1) is not chosen at random. In fact, it can be shown that L.xI n/

is an approximation to ln.1 C x/ for a finite n and x

1. The approximation

becomes exact in the limit

lim L.xI n/ D ln.1 C x/ :

n!1

Computational significance of L.xI n/

Although we can compute ln.1 C x/ on a calculator or by math.log(1+x) in

Python, you may have wondered how such a function is actually calculated in-

side the calculator or the math module. In most cases this must be done via

simple mathematical expressions such as the sum in (3.1). A calculator and the

math module will use more sophisticated formulas than (3.1) for ultimate ef-

ficiency of the calculations, but the main point is that the numerical values of

mathematical functions like ln.x/, sin.x/, and tan.x/ are usually computed by

sums similar to (3.1).

Instead of having our L function just returning the value of the sum, we could

return additional information on the error involved in the approximation of ln.1Cx/

by L.xI n/. The size of the terms decreases with increasing n, and the first neglected

term is then bigger than all the remaining terms, but not necessarily bigger than their

sum. The first neglected term is hence an indication of the size of the total error we

make, so we may use this term as a rough estimate of the error. For comparison,

we could also return the exact error since we are able to calculate the ln function by

math.log.

A new version of the L(x, n) function, where we return the value of L.xI n/,

the first neglected term, and the exact error goes as follows:

def L2(x, n):

s = 0

for i in range(1, n+1):

s += (1.0/i)\*(x/(1.0+x))\*\*i

value\_of\_sum = s

first\_neglected\_term = (1.0/(n+1))\*(x/(1.0+x))\*\*(n+1)

from math import log

exact\_error = log(1+x) - value\_of\_sum

return value\_of\_sum, first\_neglected\_term, exact\_error

# typical call:

value, approximate\_error, exact\_error = L2(x, 100)

The next section demonstrates the usage of the L2 function to judge the quality of

the approximation L.xI n/ to ln.1 C x/.

3.1.9 Functions with No Return Values

Sometimes a function just performs a set of statements, without computing objects

that are natural to return to the calling code. In such situations one can simply skip

the return statement. Some programming languages use the terms procedure or

subroutine for functions that do not return anything.102

3

Functions and Branching

Let us exemplify a function without return values by making a table of the accu-

racy of the L.xI n/ approximation to ln.1 C x/ from the previous section:

def table(x):

print ’\nx=%g, ln(1+x)=%g’ % (x, log(1+x))

for n in [1, 2, 10, 100, 500]:

value, next, error = L2(x, n)

print ’n=%-4d %-10g (next term: %8.2e ’\

’error: %8.2e)’ % (n, value, next, error)

This function just performs a set of statements that we may want to run several

times. Calling

table(10)

table(1000)

gives the output

x=10, ln(1+x)=2.3979

n=1

0.909091

(next term: 4.13e-01

n=2

1.32231

(next term: 2.50e-01

n=10

2.17907

(next term: 3.19e-02

n=100 2.39789

(next term: 6.53e-07

n=500 2.3979

(next term: 3.65e-24error: 1.49e+00)

error: 1.08e+00)

error: 2.19e-01)

error: 6.59e-06)

error: 6.22e-15)

x=1000, ln(1+x)=6.90875

n=1

0.999001

(next term: 4.99e-01

n=2

1.498

(next term: 3.32e-01

n=10

2.919

(next term: 8.99e-02

n=100 5.08989

(next term: 8.95e-03

n=500 6.34928

(next term: 1.21e-03error: 5.91e+00)

error: 5.41e+00)

error: 3.99e+00)

error: 1.82e+00)

error: 5.59e-01)

From this output we see that the sum converges much more slowly when x is large

than when x is small. We also see that the error is an order of magnitude or more

larger than the first neglected term in the sum. The functions L, L2, and table are

found in the file lnsum.py.

When there is no explicit return statement in a function, Python actually inserts

an invisible return None statement. None is a special object in Python that repre-

sents something we might think of as empty data or just “nothing”. Other computer

languages, such as C, C++, and Java, use the word void for a similar thing. Nor-

mally, one will call the table function without assigning the return value to any

variable, but if we assign the return value to a variable, result = table(500),

result will refer to a None object.

The None value is often used for variables that should exist in a program, but

where it is natural to think of the value as conceptually undefined. The standard

way to test if an object obj is set to None or not reads

if obj is None:

...

if obj is not None:

...3.1

Functions

103

One can also use obj == None. The is operator tests if two names refer to the

same object, while == tests if the contents of two objects are the same:

>>> a = 1

>>> b = a

>>> a is b

True

>>> c = 1.0

>>> a is c

False

>>> a == c

True

# a and b refer to the same object

# a and c are mathematically equal

3.1.10 Keyword Arguments

Some function arguments can be given a default value so that we may leave out

these arguments in the call. A typical function may look as

>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):

>>>

print arg1, arg2, kwarg1, kwarg2

The first two arguments, arg1 and arg2, are ordinary or positional arguments,

while the latter two are keyword arguments or named arguments. Each keyword ar-

gument has a name (in this example kwarg1 and kwarg2) and an associated default

value. The keyword arguments must always be listed after the positional arguments

in the function definition.

When calling somefunc, we may leave out some or all of the keyword argu-

ments. Keyword arguments that do not appear in the call get their values from the

specified default values. We can demonstrate the effect through some calls:

>>> somefunc(’Hello’, [1,2])

Hello [1, 2] True 0

>>> somefunc(’Hello’, [1,2], kwarg1=’Hi’)

Hello [1, 2] Hi 0

>>> somefunc(’Hello’, [1,2], kwarg2=’Hi’)

Hello [1, 2] True Hi

>>> somefunc(’Hello’, [1,2], kwarg2=’Hi’, kwarg1=6)

Hello [1, 2] 6 Hi

The sequence of the keyword arguments does not matter in the call. We may also

mix the positional and keyword arguments if we explicitly write name=value for

all arguments in the call:

>>> somefunc(kwarg2=’Hello’, arg1=’Hi’, kwarg1=6, arg2=[1,2],)

Hi [1, 2] 6 Hello

Example: Function with default parameters Consider a function of t which also

contains some parameters, here A, a, and !:

f .tI A; a; !/ D Ae at sin.!t/ :

(3.2)104

3

Functions and Branching

We can implement f as a Python function where the independent variable t is an or-

dinary positional argument, and the parameters A, a, and ! are keyword arguments

with suitable default values:

from math import pi, exp, sin

def f(t, A=1, a=1, omega=2\*pi):

return A\*exp(-a\*t)\*sin(omega\*t)

Calling f with just the t argument specified is possible:

v1 = f(0.2)

In this case we evaluate the expression e 0:2 sin.2 0:2/. Other possible calls

include

v2 = f(0.2, omega=1)

v3 = f(1, A=5, omega=pi, a=pi\*\*2)

v4 = f(A=5, a=2, t=0.01, omega=0.1)

v5 = f(0.2, 0.5, 1, 1)

You should write down the mathematical expressions that arise from these four

calls. Also observe in the third line above that a positional argument, t in that case,

can appear in between the keyword arguments if we write the positional argument

on the keyword argument form name=value. In the last line we demonstrate that

keyword arguments can be used as positional argument, i.e., the name part can be

skipped, but then the sequence of the keyword arguments in the call must match the

sequence in the function definition exactly.

Example: Computing a sum with default tolerance Consider the L.xI n/ sum

and the Python implementations L(x, n) and L2(x, n) from Sect. 3.1.8. Instead

of specifying the number of terms in the sum, n, it is better to specify a tolerance "

of the accuracy. We can use the first neglected term as an estimate of the accuracy.

This means that we sum up terms as long as the absolute value of the next term is

greater than . It is natural to provide a default value for :

def L3(x, epsilon=1.0E-6):

x = float(x)

i = 1

term = (1.0/i)\*(x/(1+x))\*\*i

s = term

while abs(term) > epsilon:

i += 1

term = (1.0/i)\*(x/(1+x))\*\*i

s += term

return s, i

Here is an example involving this function to make a table of the approximation

error as decreases:3.1

Functions

105

def table2(x):

from math import log

for k in range(4, 14, 2):

epsilon = 10\*\*(-k)

approx, n = L3(x, epsilon=epsilon)

exact = log(1+x)

exact\_error = exact - approx

The output from calling table2(10) becomes

epsilon: 1e-04, exact error: 8.18e-04, n=55

epsilon: 1e-06, exact error: 9.02e-06, n=97

epsilon: 1e-08, exact error: 8.70e-08, n=142

epsilon: 1e-10, exact error: 9.20e-10, n=187

epsilon: 1e-12, exact error: 9.31e-12, n=233

We see that the epsilon estimate is almost 10 times smaller than the exact error,

regardless of the size of epsilon. Since epsilon follows the exact error quite well

over many orders of magnitude, we may view epsilon as a useful indication of the

size of the error.

3.1.11 Doc Strings

There is a convention in Python to insert a documentation string right after the def

line of the function definition. The documentation string, known as a doc string,

should contain a short description of the purpose of the function and explain what

the different arguments and return values are. Interactive sessions from a Python

shell are also common to illustrate how the code is used. Doc strings are usually

enclosed in triple double quotes """, which allow the string to span several lines.

Here are two examples on short and long doc strings:

def C2F(C):

"""Convert Celsius degrees (C) to Fahrenheit."""

return (9.0/5)\*C + 32

def line(x0, y0, x1, y1):

"""

Compute the coefficients a and b in the mathematical

expression for a straight line y = a\*x + b that goes

through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (floats).

x1, y1: another point on the line (floats).

return: coefficients a, b (floats) for the line (y=a\*x+b).

"""

a = (y1 - y0)/float(x1 - x0)

b = y0 - a\*x0

return a, b

Note that the doc string must appear before any statement in the function body.106

3

Functions and Branching

There are several Python tools that can automatically extract doc strings from

the source code and produce various types of documentation. The leading tools is

Sphinx3 , see also [13, Appendix B.2].

The doc string can be accessed in a code as funcname.\_\_doc\_\_, where

funcname is the name of the function, e.g.,

print line.\_\_doc\_\_

which prints out the documentation of the line function above:

Compute the coefficients a and b in the mathematical

expression for a straight line y = a\*x + b that goes

through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (float objects).

x1, y1: another point on the line (float objects).

return: coefficients a, b for the line (y=a\*x+b).

If the function line is in a file funcs.py, we may also run pydoc funcs.line in

a terminal window to look the documentation of the line function in terms of the

function signature and the doc string.

Doc strings often contain interactive sessions, copied from a Python shell, to

illustrate how the function is used. We can add such a session to the doc string in

the line function:

def line(x0, y0, x1, y1):

"""

Compute the coefficients a and b in the mathematical

expression for a straight line y = a\*x + b that goes

through two points (x0,y0) and (x1,y1).

x0, y0: a point on the line (float).

x1, y1: another point on the line (float).

return: coefficients a, b (floats) for the line (y=a\*x+b).

Example:

>>> a, b = line(1, -1, 4, 3)

>>> a

1.3333333333333333

>>> b

-2.333333333333333

"""

a = (y1 - y0)/float(x1 - x0)

b = y0 - a\*x0

return a, b

A particularly nice feature is that all such interactive sessions in doc strings can

be automatically run, and new results are compared to the results found in the doc

strings. This makes it possible to use interactive sessions in doc strings both for

exemplifying how the code is used and for testing that the code works.

3

http://sphinx-doc.org/invocation.html#invocation-apidoc3.1

Functions

107

Function input and output

It is a convention in Python that function arguments represent the input data to

the function, while the returned objects represent the output data. We can sketch

a general Python function as

def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):

# modify io4, io5, io6; compute o1, o2, o3

return o1, o2, o3, io4, io5, io7

Here i1, i2, i3 are positional arguments representing input data; io4 and io5

are positional arguments representing input and output data; i6 and io7 are

keyword arguments representing input and input/output data, respectively; and

o1, o2, and o3 are computed objects in the function, representing output data

together with io4, io5, and io7. All examples later in the book will make use

of this convention.

3.1.12 Functions as Arguments to Functions

Programs doing calculus frequently need to have functions as arguments in func-

tions. For example, a mathematical function f .x/ is needed in Python functions

for

numerical root finding: solve f .x/ D 0 approximately (Sects. 4.11.2 and A.1.10)

numerical differentiation: compute f 0 .x/ approximately (Sects. B.2 and 7.3.2)

Rb

numerical integration: compute a f .x/dx approximately (Sects. B.3 and 7.3.3)

D f .x/ (Appendix E)

numerical solution of differential equations: dx

dt

In such Python functions we need to have the f .x/ function as an argument f. This

is straightforward in Python and hardly needs any explanation, but in most other

languages special constructions must be used for transferring a function to another

function as argument.

As an example, consider a function for computing the second-order derivative of

a function f .x/ numerically:

f 00 .x/

f .x h/ 2f .x/ C f .x C h/

;

h2

(3.3)

where h is a small number. The approximation (3.3) becomes exact in the limit

h ! 0. A Python function for computing (3.3) can be implemented as follows:

def diff2nd(f, x, h=1E-6):

r = (f(x-h) - 2\*f(x) + f(x+h))/float(h\*h)

return r

The f argument is like any other argument, i.e., a name for an object, here a function

object that we can call as we normally call functions. An application of diff2nd

may be108

3

Functions and Branching

def g(t):

return t\*\*(-6)

t = 1.2

d2g = diff2nd(g, t)

print "g’’(%f)=%f" % (t, d2g)

The behavior of the numerical derivative as h ! 0 From mathematics we know

that the approximation formula (3.3) becomes more accurate as h decreases. Let

us try to demonstrate this expected feature by making a table of the second-order

derivative of g.t/ D t 6 at t D 1 as h ! 0:

for k in range(1,15):

h = 10\*\*(-k)

d2g = diff2nd(g, 1, h)

print ’h=%.0e: %.5f’ % (h, d2g)

The output becomes

h=1e-01: 44.61504

h=1e-02: 42.02521

h=1e-03: 42.00025

h=1e-04: 42.00000

h=1e-05: 41.99999

h=1e-06: 42.00074

h=1e-07: 41.94423

h=1e-08: 47.73959

h=1e-09: -666.13381

h=1e-10: 0.00000

h=1e-11: 0.00000

h=1e-12: -666133814.77509

h=1e-13: 66613381477.50939

h=1e-14: 0.00000

With g.t/ D t 6 , the exact answer is g 00 .1/ D 42, but for h < 108 the computa-

tions give totally wrong answers! The problem is that for small h on a computer,

rounding errors in the formula (3.3) blow up and destroy the accuracy. The math-

ematical result that (3.3) becomes an increasingly better approximation as h gets

smaller and smaller does not hold on a computer! Or more precisely, the result

holds until h in the present case reaches 104 .

The reason for the inaccuracy is that the numerator in (3.3) for g.t/ D t 6 and

t D 1 contains subtraction of quantities that are almost equal. The result is a very

small and inaccurate number. The inaccuracy is magnified by h2 , a number that

becomes very large for small h.

Switching from the standard floating-point numbers (float) to numbers with

arbitrary high precision resolves the problem. Python has a module decimal that

can be used for this purpose. The SymPy package comes with an alternative module

mpmath, which also offers mathematical functions like sin and cos with arbitrary

precision. The file high\_precision.py solves the current problem using arith-

metics also based on the decimal and mpmath modules. With 25 digits in x and h3.1

Functions

109

inside the diff2nd function, we get accurate results for h 1013 with decimal,

while rounding errors show up for h 1010 with the mpmath module.

Nevertheless, for most practical applications of (3.3), a moderately small h, say

103 h 104 , gives sufficient accuracy and then rounding errors from float

calculations do not pose problems. Real-world science or engineering applications

usually have many parameters with uncertainty, making the end result also uncer-

tain, and formulas like (3.3) can then be computed with moderate accuracy without

affecting the overall uncertainty in the answers.

3.1.13 The Main Program

In programs containing functions we often refer to a part of the program that is

called the main program. This is the collection of all the statements outside the

functions, plus the definition of all functions. Let us look at a complete program:

from math import \*# in main

def f(x):

e = exp(-0.1\*x)

s = sin(6\*pi\*x)

return e\*s# in main

x = 2

y = f(x)

print ’f(%g)=%g’ % (x, y)# in main

# in main

# in main

The main program here consists of the lines with a comment in main. The ex-

ecution always starts with the first line in the main program. When a function is

encountered, its statements are just used to define the function – nothing gets com-

puted inside the function before we explicitly call the function, either from the main

program or from another function. All variables initialized in the main program be-

come global variables (see Sect. 3.1.3).

The program flow in the program above goes as follows:

Import functions from the math module,

define a function f(x),

define x,

call f and execute the function body,

define y as the value returned from f,

print the string.

In point 4, we jump to the f function and execute the statement inside that function

for the first time. Then we jump back to the main program and assign the float

object returned from f to the y variable.

Readers who are uncertain about the program flow and the jumps between the

main program and functions should use a debugger or the Online Python Tutor as

explained in Sect. 3.1.2.110

3

Functions and Branching

3.1.14 Lambda Functions

There is a quick one-line construction of functions that is often convenient to make

Python code compact:

f = lambda x: x\*\*2 + 4

This so-called lambda function is equivalent to writing

def f(x):

return x\*\*2 + 4

In general,

def g(arg1, arg2, arg3, ...):

return expression

can be written as

g = lambda arg1, arg2, arg3, ...: expression

Lambda functions are usually used to quickly define a function as argument to

another function. Consider, as an example, the diff2nd function from Sect. 3.1.12.

In the example from that chapter we want to differentiate g.t/ D t 6 twice and first

make a Python function g(t) and then send this g to diff2nd as argument. We can

skip the step with defining the g(t) function and instead insert a lambda function

as the f argument in the call to diff2nd:

d2 = diff2nd(lambda t: t\*\*(-6), 1, h=1E-4)

Because lambda functions saves quite some typing, at least for very small functions,

they are popular among many programmers.

Lambda functions may also take keyword arguments. For example,

d2 = diff2nd(lambda t, A=1, a=0.5: -a\*2\*t\*A\*exp(-a\*t\*\*2), 1.2)

3.2 Branching

The flow of computer programs often needs to branch. That is, if a condition is met,

we do one thing, and if not, we do another thing. A simple example is a function

defined as

(

sin x; 0 x

(3.4)

f .x/ D

0;

otherwise

In a Python implementation of this function we need to test on the value of x, which

can be done as displayed below:3.2 Branching

111

def f(x):

if 0 <= x <= pi:

value = sin(x)

else:

value = 0

return value

3.2.1 If-else Blocks

The general structure of an if-else test is

if condition:

<block of statements, executed if condition is True>

else:

<block of statements, executed if condition is False>

When condition evaluates to True, the program flow branches into the first block

of statements. If condition is False, the program flow jumps to the second block

of statements, after the else: line. As with while and for loops, the block of

statements are indented. Here is another example:

if C < -273.15:

print ’%g degrees Celsius is non-physical!’ % C

print ’The Fahrenheit temperature will not be computed.’

else:

F = 9.0/5\*C + 32

print F

print ’end of program’

The two print statements in the if block are executed if and only if C < -273.15

evaluates to True. Otherwise, we jump over the first two print statements and

carry out the computation and printing of F. The printout of end of program will

be performed regardless of the outcome of the if test since this statement is not

indented and hence neither a part of the if block nor the else block.

The else part of an if test can be skipped, if desired:

if condition:

<block of statements>

<next statement>

For example,

if C < -273.15:

print ’%s degrees Celsius is non-physical!’ % C

F = 9.0/5\*C + 32

In this case the computation of F will always be carried out, since the statement is

not indented and hence not a part of the if block.112

3

Functions and Branching

With the keyword elif, short for else if, we can have several mutually exclusive

if tests, which allows for multiple branching of the program flow:

if condition1:

<block of statements>

elif condition2:

<block of statements>

elif condition3:

<block of statements>

else:

<block of statements>

<next statement>

The last else part can be skipped if it is not needed. To illustrate multiple branch-

ing we will implement a hat function, which is widely used in advanced computer

simulations in science and industry. One example of a hat function is

8

0;

ˆ

ˆ

ˆ

<

x;

N.x/ D

ˆ

ˆ 2 x;

:̂

0;

x<0

0x<1

1x<2

x 2

(3.5)

The solid line in Fig. 5.9 in Sect. 5.4.1 illustrates the shape of this function and why

it is called a hat function. The Python implementation associated with (3.5) needs

multiple if branches:

def N(x):

if x < 0:

return 0.0

elif 0 <= x < 1:

return x

elif 1 <= x < 2:

return 2 - x

elif x >= 2:

return 0.0

This code corresponds directly to the mathematical specification, which is a sound

strategy that help reduce the amount of errors in programs. We could mention that

there is another way of constructing the if test that results in shorter code:

def N(x):

if 0 <= x < 1:

return x

elif 1 <= x < 2:

return 2 - x

else:

return 0

As a part of learning to program, understanding this latter sample code is important,

but we recommend the former solution because of its direct similarity with the

mathematical definition of the function.3.3 Mixing Loops, Branching, and Functions in Bioinformatics Examples

113

A popular programming rule is to avoid multiple return statements in a function

– there should only be one return at the end. We can do that in the N function

by introducing a local variable, assigning values to this variable in the blocks and

returning the variable at the end. However, we do not think an extra variable and an

extra line make a great improvement in such a short function. Nevertheless, in long

and complicated functions the rule can be helpful.

3.2.2

Inline if Tests

A variable is often assigned a value that depends on a boolean expression. This can

be coded using a common if-else test:

if condition:

a = value1

else:

a = value2

Because this construction is often needed, Python provides a one-line syntax for the

four lines above:

a = (value1 if condition else value2)

The parentheses are not required, but recommended style. One example is

def f(x):

return (sin(x) if 0 <= x <= 2\*pi else 0)

Since the inline if test is an expression with a value, it can be used in lambda

functions:

f = lambda x: sin(x) if 0 <= x <= 2\*pi else 0

The traditional if-else construction with indented blocks cannot be used inside

lambda functions because it is not just an expression (lambda functions cannot have

statements inside them, only a single expression).

3.3 Mixing Loops, Branching, and Functions in Bioinformatics

Examples

Life is definitely digital. The genetic code of all living organisms are represented

by a long sequence of simple molecules called nucleotides, or bases, which makes

up the Deoxyribonucleic acid, better known as DNA. There are only four such nu-

cleotides, and the entire genetic code of a human can be seen as a simple, though

3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain

increased biological understanding is much about searching in long strings for cer-

tain string patterns involving the letters A, C, G, and T. This is an integral part of114

3

Functions and Branching

bioinformatics, a scientific discipline addressing the use of computers to search for,

explore, and use information about genes, nucleic acids, and proteins.

The leading Python software for bioinformatics applications is BioPython4 . The

examples in this book (below and Sects. 6.5, 8.3.4, and 9.5) are simple illustrations

of the type of problem settings and corresponding Python implementations that are

encountered in bioinformatics. For real-world problem solving one should rather

utilize BioPython, but the sections below act as an introduction to what is inside

packages like BioPython.

We start with some very simple examples on DNA analysis that bring together

basic building blocks in programming: loops, if tests, and functions.

3.3.1 Counting Letters in DNA Strings

Given some string dna containing the letters A, C, G, or T, representing the bases

that make up DNA, we ask the question: how many times does a certain base occur

in the DNA string? For example, if dna is ATGGCATTA and we ask how many

times the base A occur in this string, the answer is 3.

A general Python implementation answering this problem can be done in many

ways. Several possible solutions are presented below.

List iteration The most straightforward solution is to loop over the letters in the

string, test if the current letter equals the desired one, and if so, increase a counter.

Looping over the letters is obvious if the letters are stored in a list. This is easily

done by converting a string to a list:

>>> list(’ATGC’)

[’A’, ’T’, ’G’, ’C’]

Our first solution becomes

def count\_v1(dna, base):

dna = list(dna) # convert string to list of letters

i = 0

# counter

for c in dna:

if c == base:

i += 1

return i

String iteration Python allows us to iterate directly over a string without convert-

ing it to a list:

>>> for c in ’ATGC’:

...

print c

A

T

G

C

4

http://biopython.org3.3 Mixing Loops, Branching, and Functions in Bioinformatics Examples

115

In fact, all built-in objects in Python that contain a set of elements in a particular

sequence allow a for loop construction of the type for element in object.

A slight improvement of our solution is therefore to iterate directly over the

string:

def count\_v2(dna, base):

i = 0 # counter

for c in dna:

if c == base:

i += 1

return i

dna = ’ATGCGGACCTAT’

base = ’C’

n = count\_v2(dna, base)

# printf-style formatting

print ’%s appears %d times in %s’ % (base, n, dna)

# or (new) format string syntax

print ’{base} appears {n} times in {dna}’.format(

base=base, n=n, dna=dna)

We have here illustrated two alternative ways of writing out text where the value

of variables are to be inserted in “slots” in the string.

Index iteration Although it is natural in Python to iterate over the letters in a string

(or more generally over elements in a sequence), programmers with experience

from other languages (Fortran, C and Java are examples) are used to for loops with

an integer counter running over all indices in a string or array:

def count\_v3(dna, base):

i = 0 # counter

for j in range(len(dna)):

if dna[j] == base:

i += 1

return i

Python indices always start at 0 so the legal indices for our string become 0,

1, . . . , len(dna)-1, where len(dna) is the number of letters in the string dna.

The range(x) function returns a list of integers 0, 1, . . . , x-1, implying that

range(len(dna)) generates all the legal indices for dna.

While loops The while loop equivalent to the last function reads

def count\_v4(dna, base):

i = 0 # counter

j = 0 # string index

while j < len(dna):

if dna[j] == base:

i += 1

j += 1

return i116

3

Functions and Branching

Correct indentation is here crucial: a typical error is to fail indenting the j += 1

line correctly.

Summing a boolean list The idea now is to create a list m where m[i] is True if

dna[i] equals the letter we search for (base). The number of True values in m is

then the number of base letters in dna. We can use the sum function to find this

number because doing arithmetics with boolean lists automatically interprets True

as 1 and False as 0. That is, sum(m) returns the number of True elements in m.

A possible function doing this is

def count\_v5(dna, base):

m = []

# matches for base in dna: m[i]=True if dna[i]==base

for c in dna:

if c == base:

m.append(True)

else:

m.append(False)

return sum(m)

Inline if test Shorter, more compact code is often a goal if the compactness en-

hances readability. The four-line if test in the previous function can be con-

densed to one line using the inline if construction: if condition value1 else

value2.

def count\_v6(dna, base):

m = []

# matches for base in dna: m[i]=True if dna[i]==base

for c in dna:

m.append(True if c == base else False)

return sum(m)

Using boolean values directly The inline if test is in fact redundant in the previ-

ous function because the value of the condition c == base can be used directly: it

has the value True or False. This saves some typing and adds clarity, at least to

Python programmers with some experience:

def count\_v7(dna, base):

m = []

# matches for base in dna: m[i]=True if dna[i]==base

for c in dna:

m.append(c == base)

return sum(m)

List comprehensions Building a list with the aid of a for loop can often be

condensed to a single line by using list comprehensions: [expr for e in

sequence], where expr is some expression normally involving the iteration

variable e. In our last example, we can introduce a list comprehension

def count\_v8(dna, base):

m = [c == base for c in dna]

return sum(m)3.3 Mixing Loops, Branching, and Functions in Bioinformatics Examples

117

Here it is tempting to get rid of the m variable and reduce the function body to

a single line:

def count\_v9(dna, base):

return sum([c == base for c in dna])

Using a sum iterator The DNA string is usually huge – 3 billion letters for the

human species. Making a boolean array with True and False values therefore

increases the memory usage by a factor of two in our sample functions count\_v5 to

count\_v9. Summing without actually storing an extra list is desirable. Fortunately,

sum([x for x in s]) can be replaced by sum(x for x in s), where the latter

sums the elements in s as x visits the elements of s one by one. Removing the

brackets therefore avoids first making a list before applying sum to that list. This is

a minor modification of the count\_v9 function:

def count\_v10(dna, base):

return sum(c == base for c in dna)

Below we shall measure the impact of the various program constructs on the

CPU time.

Extracting indices Instead of making a boolean list with elements expressing

whether a letter matches the given base or not, we may collect all the indices of the

matches. This can be done by adding an if test to the list comprehension:

def count\_v11(dna, base):

return len([i for i in range(len(dna)) if dna[i] == base])

The Online Python Tutor5 is really helpful to reach an understanding of this

compact code. Alternatively, you may play with the constructions in an interactive

Python shell:

>>> dna = ’AATGCTTA’

>>> base = ’A’

>>> indices = [i for i in range(len(dna)) if dna[i] == base]

>>> indices

[0, 1, 7]

>>> print dna[0], dna[1], dna[7] # check

A A A

Observe that the element i in the list comprehension is only made for those i where

dna[i] == base.

Using Python’s library Very often when you set out to do a task in Python, there

is already functionality for the task in the object itself, in the Python libraries, or

in third-party libraries found on the Internet. Counting how many times a letter (or

5

http://www.pythontutor.com/118

3

Functions and Branching

substring) base appears in a string dna is obviously a very common task so Python

supports it by the syntax dna.count(base):

def count\_v12(dna, base):

return dna.count(base)

def compare\_efficiency():

3.3.2 Efficiency Assessment

Now we have 11 different versions of how to count the occurrences of a letter in

a string. Which one of these implementations is the fastest? To answer the question

we need some test data, which should be a huge string dna.

Generating random DNA strings The simplest way of generating a long string is

to repeat a character a large number of times:

N = 1000000

dna = ’A’\*N

The resulting string is just ’AAA...A, of length N, which is fine for testing the effi-

ciency of Python functions. Nevertheless, it is more exciting to work with a DNA

string with letters from the whole alphabet A, C, G, and T. To make a DNA string

with a random composition of the letters we can first make a list of random letters

and then join all those letters to a string:

import random

alphabet = list(’ATGC’)

dna = [random.choice(alphabet) for i in range(N)]

dna = ’’.join(dna) # join the list elements to a string

The random.choice(x) function selects an element in the list x at random.

Note that N is very often a large number. In Python version 2.x, range(N) gen-

erates a list of N integers. We can avoid the list by using xrange which generates

an integer at a time and not the whole list. In Python version 3.x, the range func-

tion is actually the xrange function in version 2.x. Using xrange, combining the

statements, and wrapping the construction of a random DNA string in a function,

gives

import random

def generate\_string(N, alphabet=’ACGT’):

return ’’.join([random.choice(alphabet) for i in xrange(N)])

dna = generate\_string(600000)

The call generate\_string(10) may generate something like AATGGCAGAA.3.3 Mixing Loops, Branching, and Functions in Bioinformatics Examples

119

Measuring CPU time Our next goal is to see how much time the various

count\_v\* functions spend on counting letters in a huge string, which is to be

generated as shown above. Measuring the time spent in a program can be done by

the time module:

import time

...

t0 = time.clock()

# do stuff

t1 = time.clock()

cpu\_time = t1 - t0

The time.clock() function returns the CPU time spent in the program since its

start. If the interest is in the total time, also including reading and writing files,

time.time() is the appropriate function to call.

Running through all our functions made so far and recording timings can be done

by

import time

functions = [count\_v1, count\_v2, count\_v3, count\_v4,

count\_v5, count\_v6, count\_v7, count\_v8,

count\_v9, count\_v10, count\_v11, count\_v12]

timings = [] # timings[i] holds CPU time for functions[i]

for function in functions:

t0 = time.clock()

function(dna, ’A’)

t1 = time.clock()

cpu\_time = t1 - t0

timings.append(cpu\_time)

In Python, functions are ordinary objects so making a list of functions is no more

special than making a list of strings or numbers.

We can now iterate over timings and functions simultaneously via zip to

make a nice printout of the results:

for cpu\_time, function in zip(timings, functions):

print ’{f:<9s}: {cpu:.2f} s’.format(

f=function.func\_name, cpu=cpu\_time)

Timings on a MacBook Air 11 running Ubuntu show that the functions using

list.append require almost the double of the time of the functions that work with

list comprehensions. Even faster is the simple iteration over the string. However,

the built-in count functionality of strings (dna.count(base)) runs over 30 times

faster than the best of our handwritten Python functions! The reason is that the for

loop needed to count in dna.count(base) is actually implemented in C and runs

very much faster than loops in Python.

A clear lesson learned is: google around before you start out to implement what

seems to be a quite common task. Others have probably already done it for you, and

most likely is their solution much better than what you can (easily) come up with.120

3

Functions and Branching

3.3.3 Verifying the Implementations

We end this section with showing how to make tests that verify our 12 counting

functions. To this end, we make a new function that first computes a certainly

correct answer to a counting problem and then calls all the count\_\* functions,

stored in the list functions, to check that each call has the correct result:

def test\_count\_all():

dna = ’ATTTGCGGTCCAAA’

exact = dna.count(’A’)

for f in functions:

if f(dna, ’A’) != exact:

print f.\_\_name\_\_, ’failed’

Here, we believe in dna.count(’A’) as the correct answer.

We might take this test function one step further and adopt the conventions in

the pytest6 and nose7 testing frameworks for Python code. (See Sect. H.9 for more

information about pytest and nose.)

These conventions say that the test function should

have a name starting with test\_;

have no arguments;

let a boolean variable, say success, be True if a test passes and be False if the

test fails;

create a message about what failed, stored in some string, say msg;

use the construction assert success, msg, which will abort the program and

write out the error message msg if success is False.

The pytest and nose test frameworks can search for all Python files in a folder

tree, run all test\_\*() functions, and report how many of the tests that failed, if we

adopt the conventions above. Our revised test function becomes

def test\_count\_all():

dna = ’ATTTGCGGTCCAAA’

expected = dna.count(’A’)

functions = [count\_v1, count\_v2, count\_v3, count\_v4,

count\_v5, count\_v6, count\_v7, count\_v8,

count\_v9, count\_v10, count\_v11, count\_v12]

for f in functions:

success = f(dna, ’A’) == expected

msg = ’%s failed’ % f.\_\_name\_\_

assert success, msg

It is worth notifying that the name of a function f, as a string object, is given by

f.\_\_name\_\_, and we make use of this information to construct an informative mes-

sage in case a test fails.

6

7

http://pytest.org

https://nose.readthedocs.org3.4 Summary

121

It is a good habit to write such test functions since the execution of all tests in all

files can be fully automated. Every time you do a change in some file you can with

minimum effort rerun all tests.

The entire suite of functions presented above, including the timings and tests,

can be found in the file count.py.

3.4 Summary

3.4.1 Chapter Topics

User-defined functions Functions are useful (i) when a set of commands are to

be executed several times, or (ii) to partition the program into smaller pieces to

gain better overview. Function arguments are local variables inside the function

whose values are set when calling the function. Remember that when you write

the function, the values of the arguments are not known. Here is an example of

a function for polynomials of 2nd degree:

# function definition:

def quadratic\_polynomial(x, a, b, c)

value = a\*x\*x + b\*x + c

derivative = 2\*a\*x + b

return value, derivative

# function call:

x = 1

p, dp = quadratic\_polynomial(x, 2, 0.5, 1)

p, dp = quadratic\_polynomial(x=x, a=-4, b=0.5, c=0)

The sequence of the arguments is important, unless all arguments are given as

name=value.

Functions may have no arguments and/or no return value(s):

def print\_date():

"""Print the current date in the format ’Jan 07, 2007’."""

import time

print time.strftime("%b %d, %Y")

# call:

print\_date()

A common error is to forget the parentheses: print\_date is the function object

itself, while print\_date() is a call to the function.

Keyword arguments Function arguments with default values are called keyword

arguments, and they help to document the meaning of arguments in function calls.

They also make it possible to specify just a subset of the arguments in function calls.122

3

Functions and Branching

from math import exp, sin, pi

def f(x, A=1, a=1, w=pi):

return A\*exp(-a\*x)\*sin(w\*x)

f1 = f(0)

x2 = 0.1

f2 = f(x2, w=2\*pi)

f3 = f(x2, w=4\*pi, A=10, a=0.1)

f4 = f(w=4\*pi, A=10, a=0.1, x=x2)

The sequence of the keyword arguments can be arbitrary, and the keyword argu-

ments that are not listed in the call get their default values according to the function

definition. The non-keyword arguments are called positional arguments, which is

x in this example. Positional arguments must be listed before the keyword argu-

ments. However, also a positional argument can appear as name=value in the call

(see the last line above), and this syntax allows any positional argument to be listed

anywhere in the call.

If tests The if-elif-else tests are used to branch the flow of statements. That

is, different sets of statements are executed depending on whether some conditions

are True or False.

def f(x):

if x < 0:

value = -1

elif x >= 0 and x <= 1:

value = x

else:

value = 1

return value

Inline if tests Assigning a variable one value if a condition is True and another

value if False, is compactly done with an inline if test:

sign = -1 if a < 0 else 1

Terminology The important computer science terms in this chapter are

function

method

return statement

positional arguments

keyword arguments

local and global variables

doc strings

if tests with if, elif, and else (branching)

the None object

test functions (for verification)3.4 Summary

123

3.4.2 Example: Numerical Integration

Problem An integral

Zb

f .x/dx

a

can be approximated by the so-called Simpson’s rule:

1

0

n=2

n=21

X

X

ba @

f .a C .2i 1/h/ C 2

f .a C 2ih/A : (3.6)

f .a/ C f .b/ C 4

3n

i D1

i D1

Here, h D .b a/=n and n must be an even integer. The problem is to make

a function Simpson(f, a, b, n=500) that returns the right-hand side formula of

(3.6). To verify the implementation, one can make use of the fact that Simpson’s

rule is exact for all

R polynomials f .x/ of degree 2. Apply the Simpson function

to the integral 32 0 sin3 xdx, which has exact value 2, and investigate how the

approximation error varies with n.

Solution The evaluation of the formula

P(3.6) in a program is straightforward if we

know how to implement summation ( ) and how to call f . A Python recipe for

P

calculating sums is given in Sect. 3.1.8. Basically, N

i DM q.i/, for some expression

q.i/ involving i, is coded with the aid of a for loop over i and an accumulation

variable s for building up the sum, one term at a time:

s = 0

for i in range(M, N+1):

s += q(i)

The Simpson function can then be coded as

def Simpson(f, a, b, n=500):

h = (b - a)/float(n)

sum1 = 0

for i in range(1, n/2 + 1):

sum1 += f(a + (2\*i-1)\*h)

sum2 = 0

for i in range(1, n/2):

sum2 += f(a + 2\*i\*h)

integral = (b-a)/(3\*n)\*(f(a) + f(b) + 4\*sum1 + 2\*sum2)

return integral

Note that Simpson can integrate any Python function f of one variable. Specifically,

we can implement

3

h.x/ D sin3 xdx

2124

3

Functions and Branching

in a Python function

def h(x):

return (3./2)\*sin(x)\*\*3

R

and call Simpson to compute 0 h.x/dx for various choices of n, as requested:

from math import sin, pi

def application():

print ’Integral of 1.5\*sin^3 from 0 to pi:’

for n in 2, 6, 12, 100, 500:

approx = Simpson(h, 0, pi, n)

print ’n=%3d, approx=%18.15f, error=%9.2E’ % \

(n, approx, 2-approx)

(We have put the statements inside a function, here called application, mainly

to group them, and not because application will be called several times or with

different arguments.)

Verification Calling application() leads to the output

Integral of 1.5\*sin^3 from 0 to pi:

n= 2, approx= 3.141592653589793, error=-1.14E+00

n= 6, approx= 1.989171700583579, error= 1.08E-02

n= 12, approx= 1.999489233010781, error= 5.11E-04

n=100, approx= 1.999999902476350, error= 9.75E-08

n=500, approx= 1.999999999844138, error= 1.56E-10

We clearly see that the approximation improves as n increases. However, every

computation will give an answer that deviates from the exact value 2. We cannot

from this test alone know if the errors above are those implied by the approximation

only, or if there are additional programming mistakes.

A much better way of verifying the implementation is therefore to look for test

cases where the numerical approximation formula is exact, such that we know ex-

actly what the result of the function should be. Since it is stated that the formula is

exact for polynomials up to second degree, we just test the Simpson function on an

“arbitrary” parabola, say

Z2

3x 2 7x C 2:5 dx :

3=2

This integral equals G.2/ G.3=2/, where G.x/ D x 3 3:5x 2 C 2:5x.

A fundamental problem arises if we compare the computed integral value with

the exact result using ==, because rounding errors may lead to small differences and

hence a false equality test. Consider the simple problem3.4 Summary

125

>>> 0.1 + 0.2 == 0.3

False

>>> 0.1 + 0.2

0.30000000000000004

We see that 0.1 + 0.2 leads to a small error in the 17th decimal place. We must

therefore make comparisons with tolerances:

>>> tol = 1E-14

>>> abs(0.3 - (0.1 + 0.2)) < tol

True

In this particular example,

>>> abs(0.3 - (0.1 + 0.2))

5.551115123125783e-17

so a tolerance of 1016 would work, but in algorithms with many more arithmetic

operations, rounding errors may accumulate so 1015 or 1014 are more appropriate

tolerances.

The following implementation of a test function applies a tolerance in the test

for equality:

def g(x):

return 3\*x\*\*2 - 7\*x + 2.5

def G(x):

return x\*\*3 - 3.5\*x\*\*2 + 2.5\*x

def test\_Simpson():

a = 1.5

b = 2.0

n = 8

expact = G(b) - G(a)

approx = Simpson(g, a, b, n)

success = abs(exact - approx) < 1E-14

if not success:

print ’Error: wrong integral of quadratic function’

The g and G functions are only of interest inside the test\_Simpson function.

Many think the code becomes easier to read and understand if g and G are moved

inside test\_Simpson, which is indeed possible in Python:

def test\_Simpson():

def g(x):

# test function for exact integration by Simpson’s rule

return 3\*x\*\*2 - 7\*x + 2.5

def G(x):

# integral of g(x)

return x\*\*3 - 3.5\*x\*\*2 + 2.5\*x126

3

Functions and Branching

a = 1.5

b = 2.0

n = 8

exact = G(b) - G(a)

approx = Simpson(g, a, b, n)

success = abs(exact - approx) < 1E-14

if not success:

print ’Error: cannot integrate a quadratic function exactly’

We shall make it a habit to write functions like test\_Simpson for verifying

implementations.

Unit tests and test functions When testing software for correctness, it is consid-

ered good practice to break the software into units and test the behavior of each unit.

This is called unit testing. In scientific computing, one unit is often a numerical al-

gorithm, like Simpson’s method. The testing of a unit is performed with a dedicated

test function.

As was mentioned in Sect. 3.3.3, it can be wise to write test functions accord-

ing to the conventions needed for applying the pytest and nose testing frameworks

(Sect. H.9). Our pytest/nose-compatible test function then looks as follows:

def test\_Simpson():

"""Test exact integration of quadratic polynomials."""

a = 1.5

b = 2.0

n = 8

g = lambda x: 3\*x\*\*2 - 7\*x + 2.5

# test integrand

G = lambda x: x\*\*3 - 3.5\*x\*\*2 + 2.5\*x # integral of g

exact = G(b) - G(a)

approx = Simpson(g, a, b, n)

success = abs(exact - approx) < 1E-14

msg = ’Simpson: %g, exact: %g’ % (approx, exact)

assert success, msg

Here we have also made the test function more compact by utilizing lambda func-

tions for g and G (see Sect. 3.1.14).

Checking the validity of function arguments Another improvement is to increase

the robustness of the function. That is, to check that the input data, i.e., the argu-

ments, are acceptable. Here we may test if b > a and if n is an even integer. For the

latter test, we make use of the mod function: mod(n, d ) gives the remainder when

n is divided by d (both n and d are integers). Mathematically, if p is the largest

integer such that pd n, then mod(n, d ) is n pd . For example, mod(3, 2) is 1,

mod(3, 1) is 0, mod (3, 3) is 0, and mod(18, 8) is 2. The point is that n divided by

d is an integer when mod(n, d ) is zero. In Python, the percentage sign is used for

the mod function:

>>> 18 % 8

23.5 Exercises

127

To test if n is an odd integer, we see if it can be divided by 2 and yield an integer

without any reminder: n % 2 == 0.

The improved Simpson function with validity tests on the provided arguments,

as well as a doc string (Sect. 3.1.11), can look like this:

def Simpson(f, a, b, n=500):

"""

Return the approximation of the integral of f

from a to b using Simpson’s rule with n intervals.

"""

if a > b:

print ’Error: a=%g > b=%g’ % (a, b)

return None

# check that n is even:

if n % 2 != 0:

print ’Error: n=%d is not an even integer!’ % n

n = n+1 # make n even

h = (b - a)/float(n)

sum1 = 0

for i in range(1, n/2 + 1):

sum1 += f(a + (2\*i-1)\*h)

sum2 = 0

for i in range(1, n/2):

sum2 += f(a + 2\*i\*h)

integral = (b-a)/(3\*n)\*(f(a) + f(b) + 4\*sum1 + 2\*sum2)

return integral

The complete code is found in the file Simpson.py.

A very good exercise is to simulate the program flow by hand, starting with the

call to the application function. The Online Python Tutor8 or a debugger (see

Sect. F.1) are convenient tools for controlling that your thinking is correct.

3.5 Exercises

Exercise 3.1: Implement a simple mathematical function

Implement the mathematical function

g.t/ D e t sin. t/;

in a Python function g(t). Print out g.0/ and g.1/.

Filename: expsin.

Exercise 3.2: Implement a simple mathematical function with a parameter

Let us extend the function g.t/ in Exercise 3.1 to

h.t/ D e at sin. t/;

8

http://www.pythontutor.com/128

3

Functions and Branching

where a is a parameter. How can the corresponding Python function be imple-

mented in this case? Print out h.0/ and h.1/ for the case a D 10.

Filename: expsin\_a.

Exercise 3.3: Explain how a program works

Explain how the following program works:

def add(A, B):

C = A + B

return C

a = 3

b = 2

print add(a, b)

print add(2\*a, b+1)\*3

Figure out what is being printed without running the program.

Filename: explain\_func.

Exercise 3.4: Write a Fahrenheit-Celsius conversion functions

The formula for converting Fahrenheit degrees to Celsius reads

C D

5

.F 32/ :

9

(3.7)

Write a function C(F) that implements this formula. Also write the inverse function

F(C) for going from Celsius to Fahrenheit degrees. How can you test that the two

functions work?

Hint C(F(c)) should result in c and F(C(f)) should result in f.

Filename: f2c.

Exercise 3.5: Write a test function for Exercise 3.4

Write a test function test\_F\_C that checks the computation of C(F(c)) and

F(C(f)), involving the C(F) and F(C) functions in Exercise 3.4.

Hint Use a tolerance in the comparison. Let the test function follow the conven-

tions in the nose and pytest frameworks (see Sect. 3.3.3 for a first intro and Sect. H.9

for more overview).

Filename: test\_f2c.

Exercise 3.6: Given a test function, write the function

Here is a test function:

def test\_double():

assert double(2) == 4

assert abs(double(0.1) - 0.2) < 1E-15

assert double([1, 2]) == [1, 2, 1, 2]

assert double((1, 2)) == (1, 2, 1, 2)

assert double(3+4j) == 6+8j

assert double(’hello’) == ’hellohello’3.5 Exercises

129

Write the associated function to be tested (double) and run test\_double.

Filename: test\_double.

Exercise 3.7: Evaluate a sum and write a test function

P

1

a) Write a Python function sum\_1k(M) that returns the sum s D M

kD1 k .

b) Compute s for the case M D 3 by hand and write another function test\_sum\_

1k() that calls sum\_1k(3) and checks that the answer is correct.

Hint We recommend that test\_sum\_1k follows the conventions of the pytest and

nose testing frameworks as explained in Sects. 3.3.3 and 3.4.2 (see also Sect. H.9).

Filename: sum\_func.

Exercise 3.8: Write a function for solving ax 2 C bx C c D 0

a) Given a quadratic equation ax 2 C bx C c D 0, write a function roots(a,

b, c) that returns the two roots of the equation. The returned roots should be

float-type objects when the roots are real, otherwise complex-type objects.

Hint You can test on the sign of the expression in the square root and return stan-

dard float or complex Python objects accordingly. Alternatively, you can simply

use sqrt from the numpy.lib.scimath library, see Chap. 1.6.3. This sqrt func-

tion returns an object of type numpy.complex128 in case of a negative argument

(and hence a complex square root) and an object of type numpy.float64 other-

wise.

b) Construct two test cases with known solutions, one with real roots and the

other with complex roots. Implement the two test cases in two test functions

test\_roots\_float and test\_roots\_complex, where you call the roots

function and check the value of the returned objects.

Filename: roots\_quadratic.

Exercise 3.9: Implement the sum function

The standard Python function called sum takes a list as argument and computes the

sum of the elements in the list:

>>> sum([1,3,5,-5])

4

>>> sum([[1,2], [4,3], [8,1]])

[1, 2, 4, 3, 8, 1]

>>> sum([’Hello, ’, ’World!’])

’Hello, World!’

Implement your own version of sum.

Filename: mysum.130

3

Functions and Branching

Exercise 3.10: Compute a polynomial via a product

Given n C 1 roots r0 ; r1 ; : : : ; rn of a polynomial p.x/ of degree n C 1, p.x/ can be

computed by

p.x/ D

n

Y

.x ri / D .x r0 /.x r1 / .x rn1 /.x rn / :

(3.8)

i D0

Write a function poly(x, roots) that takes x and a list roots of the roots as

arguments and returns p.x/. Construct a test case for verifying the implementation.

Filename: polyprod.

Exercise 3.11: Integrate a function by the Trapezoidal rule

a) An approximation to the integral of a function f .x/ over an interval Œa; b can

be found by first approximating f .x/ by the straight line that goes through the

end points .a; f .a// and .b; f .b//, and then finding the area under the straight

line, which is the area of a trapezoid. The resulting formula becomes

Zb

f .x/dx

ba

.f .a/ C f .b// :

2

(3.9)

a

Write a function trapezint1(f, a, b) that returns this approximation to the

integral. The argument f is a Python implementation f(x) of the mathematical

function f .x/.

R

b) Use the approximation (3.9) to compute the following integrals: 0 cos x dx,

R

R =2

sin x dx, In each case, write out the error, i.e., the dif-

0 sin x dx, and 0

ference between the exact integral and the approximation (3.9). Make rough

sketches of the trapezoid for each integral in order to understand how the method

behaves in the different cases.

c) We can easily improve the formula (3.9) by approximating the area under the

function f .x/ by two equal-sized trapezoids. Derive a formula for this ap-

proximation and implement it in a function trapezint2(f, a, b). Run the

examples from b) and see how much better the new formula is. Make sketches

of the two trapezoids in each case.

d) A further improvement of the approximate integration method from c) is to di-

vide the area under the f .x/ curve into n equal-sized trapezoids. Based on this

idea, derive the following formula for approximating the integral:

Zb

f .x/dx

a

n1

X

1

i D1

2

h .f .xi / C f .xi C1 // ;

(3.10)

where h is the width of the trapezoids, h D .b a/=n, and xi D a C ih,

i D 0; : : : ; n, are the coordinates of the sides of the trapezoids. The figure

below visualizes the idea of the Trapezoidal rule.3.5 Exercises

131

Implement (3.10) in a Python function trapezint(f, a, b, n). Run the

examples from b) with n D 10.

e) Write a test function test\_trapezint() for verifying the implementation of

the function trapezint in d).

Hint Obviously, the Trapezoidal method integrates linear functions exactly for

any n. Another more surprising result is that the method is also exact for, e.g.,

R 2

0 cos x dx for any n. Use one of these cases for the test function test\_trapezint.

Filename: trapezint.

Remarks Formula (3.10) is not the most common way of expressing the Trape-

zoidal integration rule. The reason is that f .xi C1 / is evaluated twice, first in term

i and then as f .xi / in term i C 1. The formula can be further developed to avoid

unnecessary evaluations of f .xi C1 /, which results in the standard form

Zb

f .x/dx

a

n1

X

1

f .xi / :

h.f .a/ C f .b// C h

2

i D1

(3.11)

Exercise 3.12: Derive the general Midpoint integration rule

The idea of the Midpoint rule for integration is to divide the area under the curve

f .x/ into n equal-sized rectangles (instead of trapezoids as in Exercise 3.11). The

height of the rectangle is determined by the value of f at the midpoint of the rect-

angle. The figure below illustrates the idea.132

3

Functions and Branching

Compute the area of each rectangle, sum them up, and arrive at the formula for

the Midpoint rule:

Zb

f .x/dx h

a

n1

X

1

f .a C ih C h/;

2

i D0

(3.12)

where h D .b a/=n is the width of each rectangle. Implement this formula in

a Python function midpointint(f, a, b, n) and test the function on the exam-

ples listed in Exercise 3.11b. How do the errors in the Midpoint rule compare with

those of the Trapezoidal rule for n D 1 and n D 10?

Filename: midpointint.

Exercise 3.13: Make an adaptive Trapezoidal rule

A problem with the Trapezoidal integration rule (3.10) in Exercise 3.11 is to decide

how many trapezoids (n) to use in order to achieve a desired accuracy. Let E be the

error in the Trapezoidal method, i.e., the difference between the exact integral and

that produced by (3.10). We would like to prescribe a (small) tolerance and find

an n such that E . R

b

Since the exact value a f .x/dx is not available (that is why we use a numer-

ical method!), it is challenging to compute E. Nevertheless, it has been shown by

mathematicians that

E

1

.b a/h2 max jf 00 .x/j :

x2Œa;b

12

(3.13)

The maximum of jf 00 .x/j can be computed (approximately) by evaluating f 00 .x/

at a large number of points in Œa; b, taking the absolute value jf 00 .x/j, and finding3.5 Exercises

133

the maximum value of these. The double derivative can be computed by a finite

difference formula:

f 00 .x/

f .x C h/ 2f .x/ C f .x h/

:

h2

With the computed estimate of max jf 00 .x/j we can find h from setting the right-

hand side in (3.13) equal to the desired tolerance:

1

.b a/h2 max jf 00 .x/j D :

x2Œa;b

12

Solving with respect to h gives

hD

p

1=2

00

12 .b a/ max jf .x/j

:

x2Œa;b

(3.14)

With n D .b a/= h we have the n that corresponds to the desired accuracy .

a) Make a Python function adaptive\_trapezint(f, a, b, eps=1E-5) for

Rb

computing the integral a f .x/dx with an error less than or equal to (eps).

Hint Compute the n corresponding to as explained above and call trapezint(f,

a, b, n) from Exercise 3.11.

b) Apply the function to compute the integrals from Exercise 3.11b. Write out the

exact error and the estimated n for each case.

Filename: adaptive\_trapezint.

Remarks A numerical method that applies an expression for the error to adapt the

choice of the discretization parameter to a desired error tolerance, is known as an

adaptive numerical method. The advantage of an adaptive method is that one can

control the approximation error, and there is no need for the user to determine an

appropriate number of intervals n.

Exercise 3.14: Simulate a program by hand

Simulate the following program by hand to explain what is printed.

def a(x):

q = 2

x = 3\*x

return q + x

def b(x):

global q

q += x

return q + x

q = 0

x = 3

print a(x), b(x), a(x), b(x)134

3

Functions and Branching

Hint If you encounter problems with understanding function calls and local versus

global variables, paste the code into the Online Python Tutor9 and step through the

code to get a better explanation of what happens.

Filename: simulate\_func.

Exercise 3.15: Debug a given test function

Given a Python function

def triple(x):

return x + x\*2

we want to test it with a proper test function. The following function is written

def test\_triple():

assert triple(3) == 9

assert triple(0.1) == 0.3

assert triple([1, 2]) == [1, 2, 1, 2, 1, 2]

assert triple(’hello ’) == ’hello hello 2’

What is wrong with the test function? Write a test function where all boolean

comparisons work well.

Filename: test\_triple.

Exercise 3.16: Compute the area of an arbitrary triangle

An arbitrary triangle can be described by the coordinates of its three vertices:

.x1 ; y1 /, .x2 ; y2 /, .x3 ; y3 /, numbered in a counterclockwise direction. The area of

the triangle is given by the formula

1

(3.15)

jx2 y3 x3 y2 x1 y3 C x3 y1 C x1 y2 x2 y1 j :

2

Write a function triangle\_area(vertices) that returns the area of a triangle

whose vertices are specified by the argument vertices, which is a nested list of

the vertex coordinates. Make sure your implementation passes the following test

function, which also illustrates how the triangle\_area function works:

AD

def test\_triangle\_area():

"""

Verify the area of a triangle with vertex coordinates

(0,0), (1,0), and (0,2).

"""

v1 = (0,0); v2 = (1,0); v3 = (0,2)

vertices = [v1, v2, v3]

expected = 1

computed = triangle\_area(vertices)

tol = 1E-14

success = diff(expected - computed) < tol

msg = ’computed area=%g != %g (expected)’ % \

(computed, expected)

assert success, msg

Filename: area\_triangle.

9

http://www.pythontutor.com/visualize.html3.5 Exercises

135

Exercise 3.17: Compute the length of a path

Some object is moving along a path in the plane. At n C 1 points of time we have

recorded the corresponding .x; y/ positions of the object: .x0 ; y0 /, .x1 ; y2 /, : : :,

.xn ; yn /. The total length L of the path from .x0 ; y0 / to .xn ; yn / is the sum of all

the individual line segments (.xi 1 ; yi 1 / to .xi ; yi /, i D 1; : : : ; n):

LD

n p

X

.xi xi 1 /2 C .yi yi 1 /2 :

(3.16)

i D1

a) Make a Python function pathlength(x, y) for computing L according to the

formula. The arguments x and y hold all the x0 ; : : : ; xn and y0 ; : : : ; yn coordi-

nates, respectively.

b) Write a test function test\_pathlength() where you check that pathlength

returns the correct length in a test problem.

Filename: pathlength.

Exercise 3.18: Approximate

The value of equals the circumference of a circle with radius 1=2. Suppose we

approximate the circumference by a polygon through n C 1 points on the circle.

The length of this polygon can be found using the pathlength function from Ex-

ercise 3.17. Compute n C 1 points .xi ; yi / along a circle with radius 1=2 according

to the formulas

xi D

1

cos.2 i=n/;

2

yi D

1

sin.2 i=n/;

2

i D 0; : : : ; n :

Call the pathlength function and write out the error in the approximation of for

n D 2k , k D 2; 3; : : : ; 10.

Filename: pi\_approx.

Exercise 3.19: Compute the area of a polygon

One of the most important mathematical problems through all times has been to

find the area of a polygon. For example, real estate areas often had the shape of

polygons, and the tax was proportional to the area. Suppose we have some poly-

gon with vertices (“corners”) specified by the coordinates .x1 ; y1 /, .x2 ; y2 /, : : :,

.xn ; yn /, numbered either in a clockwise or counter clockwise fashion around the

polygon. The area A of the polygon can amazingly be computed by just knowing

the boundary coordinates:

AD

1

j.x1 y2 C x2 y3 C C xn1 yn C xn y1 /

2

.y1 x2 C y2 x3 C C yn1 xn C yn x1 /j :

(3.17)

Write a function polygon\_area(x, y) that takes two coordinate lists with the

vertices as arguments and returns the area.

Test the function on a triangle, a quadrilateral, and a pentagon where you can

calculate the area by alternative methods for comparison.136

3

Functions and Branching

Hint Since Python lists and arrays has 0 as their first index, it is wise to rewrite the

mathematical formula in terms of vertex coordinates numbered as x0 ; x1 ; : : : ; xn1

and y0 ; y1 ; : : : ; yn1 before you start programming.

Filename: polygon\_area.

Exercise 3.20: Write functions

Three functions, hw1, hw2, and hw3, work as follows:

>>> print hw1()

Hello, World!

>>> hw2()

Hello, World!

>>> print hw3(’Hello, ’, ’World!’)

Hello, World!

>>> print hw3(’Python ’, ’function’)

Python function

Write the three functions.

Filename: hw\_func.

Exercise 3.21: Approximate a function by a sum of sines

We consider the piecewise constant function

8

ˆ

0 < t < T =2;

< 1;

f .t/ D

0;

t D T =2;

:̂

1; T =2 < t < T

(3.18)

Sketch this function on a piece of paper. One can approximate f .t/ by the sum

S.tI n/ D

n

2.2i 1/ t

4X 1

:

sin

i D1 2i 1

T

(3.19)

It can be shown that S.tI n/ ! f .t/ as n ! 1.

a) Write a Python function S(t, n, T) for returning the value of S.tI n/.

b) Write a Python function f(t, T) for computing f .t/.

c) Write out tabular information showing how the error f .t/ S.tI n/ varies with

n and t for the cases where n D 1; 3; 5; 10; 30; 100 and t D ˛T , with T D 2,

and ˛ D 0:01; 0:25; 0:49. Use the table to comment on how the quality of the

approximation depends on ˛ and n.

Filename: sinesum1.

Remarks A sum of sine and/or cosine functions, as in (3.19), is called a Fourier

series. Approximating a function by a Fourier series is a very important technique

in science and technology. Exercise 5.41 asks for visualization of how well S.tI n/

approximates f .t/ for some values of n.3.5 Exercises

137

Exercise 3.22: Implement a Gaussian function

Make a Python function gauss(x, m=0, s=1) for computing the Gaussian func-

tion

1

1 x m 2

f .x/ D p

exp

:

2

s

2 s

Write out a nicely formatted table of x and f .x/ values for n uniformly spaced x

values in Œm 5s; m C 5s. (Choose m, s, and n as you like.)

Filename: gaussian2.

Exercise 3.23: Wrap a formula in a function

Implement the formula (1.9) from Exercise 1.12 in a Python function with three

arguments: egg(M, To=20, Ty=70). The parameters , K, c, and Tw can be set

as local (constant) variables inside the function. Let t be returned from the function.

Compute t for a soft and hard boiled egg, of a small (M D 47 g) and large (M D 67

g) size, taken from the fridge (To D 4 C) and from a hot room (To D 25 C).

Filename: egg\_func.

Exercise 3.24: Write a function for numerical differentiation

The formula

f .x C h/ f .x h/

f 0 .x/

(3.20)

2h

can be used to find an approximate derivative of a mathematical function f .x/ if h

is small.

a) Write a function diff(f, x, h=1E-5) that returns the approximation (3.20)

of the derivative of a mathematical function represented by a Python function

f(x).

b) Write a function test\_diff() that verifies the implementation of the function

diff. As test case, one can use the fact that (3.20) is exact for quadratic func-

tions (at least for not so small h values that rounding errors in (3.20) become

significant – you have to experiment with finding a suitable tolerance and h).

Follow the conventions of the pytest and nose testing frameworks, as outlined in

Exercise 3.7 and Sects. 3.3.3, 3.4.2, and H.9.

c) Apply (3.20) to differentiate

f .x/ D e x at x D 0,

2

f .x/ D e 2x at x D 0,

f .x/ D cos x at x D 2,

f .x/ D ln x at x D 1.

Use h D 0:01. In each case, write out the error, i.e., the difference between

the exact derivative and the result of (3.20). Collect these four examples in

a function application().

Filename: centered\_diff.

Exercise 3.25: Implement the factorial function

The factorial of n is written as nŠ and defined as

nŠ D n.n 1/.n 2/ 2 1;

(3.21)138

3

Functions and Branching

with the special cases

1Š D 1;

0Š D 1 :

(3.22)

For example, 4Š D 4 3 2 1 D 24, and 2Š D 2 1 D 2. Write a Python

function fact(n) that returns nŠ. (Do not simply call the ready-made function

math.factorial(n) – that is considered cheating in this context!)

Make sure your fact function passes the test in the following test function:

def test\_fact():

# Check an arbitrary case

n = 4

expected = 4\*3\*2\*1

computed = fact(n)

assert computed == expected

# Check the special cases

assert fact(0) == 1

assert fact(1) == 1

Hint Return 1 immediately if x is 1 or 0, otherwise use a loop to compute nŠ.

Filename: fact.

Exercise 3.26: Compute velocity and acceleration from 1D position data

Suppose we have recorded GPS coordinates x0 ; : : : ; xn at times t0 ; : : : ; tn while

running or driving along a straight road. We want to compute the velocity vi and

acceleration ai from these position coordinates. Using finite difference approxima-

tions, one can establish the formulas

vi

xi C1 xi 1

;

ti C1 ti 1

ai 2.ti C1 ti 1 /1

(3.23)

xi C1 xi

xi xi 1

ti C1 ti

ti ti 1

;

(3.24)

for i D 1; : : : ; n 1 (vi and ai correspond to the velocity and acceleration at point

xi at time ti , respectively).

a) Write a Python function kinematics(i, x, t) for computing vi and ai ,

given the arrays x and t of position and time coordinates (x0 ; : : : ; xn and

t0 ; : : : ; tn ).

b) Write a Python function test\_kinematics() for testing the implementation

in the case of constant velocity V . Set t0 D 0, t1 D 0:5, t2 D 1:5, and t3 D 2:2,

and xi D V ti . Call the kinematics function for the legal i values.

Filename: kinematics1.

Exercise 3.27: Find the max and min values of a function

The maximum and minimum values of a mathematical function f .x/ on Œa; b can

be found by computing f at a large number (n) of points and selecting the maxi-

mum and minimum values at these points. Write a Python function maxmin(f, a,

b, n=1000) that returns the maximum and minimum value of a function f(x).

Also write a test function for verifying the implementation for f .x/ D cos x,

x 2 Œ=2; 2.3.5 Exercises

139

Hint The x points where the mathematical function is to be evaluated can be uni-

formly distributed: xi D a C ih, i D 0; : : : ; n 1, h D .b a/=.n 1/. The

Python functions max(y) and min(y) return the maximum and minimum values in

the list y, respectively.

Filename: maxmin\_f.

Exercise 3.28: Find the max and min elements in a list

Given a list a, the max function in Python’s standard library computes the largest

element in a: max(a). Similarly, min(a) returns the smallest element in a. Write

your own max and min functions.

Hint Initialize a variable max\_elem by the first element in the list, then visit all the

remaining elements (a[1:]), compare each element to max\_elem, and if greater, set

max\_elem equal to that element. Use a similar technique to compute the minimum

element.

Filename: maxmin\_list.

Exercise 3.29: Implement the Heaviside function

The following step function is known as the Heaviside function and is widely used

in mathematics:

(

0; x < 0

(3.25)

H.x/ D

1; x 0

a) Implement H.x/ in a Python function H(x).

b) Make a Python function test\_H() for testing the implementation of H(x).

Compute H.10/, H.1015 /, H.0/, H.1015 /, H.10/ and test that the an-

swers are correct.

Filename: Heaviside.

Exercise 3.30: Implement a smoothed Heaviside function

The Heaviside function (3.25) listed in Exercise 3.29 is discontinuous. It is in many

numerical applications advantageous to work with a smooth version of the Heavi-

side function where the function itself and its first derivative are continuous. One

such smoothed Heaviside function is

8

ˆ

x < ;

< 0;

x

1

x

1

(3.26)

H .x/ D

2 C 2 C 2 sin ; x

:̂

1;

x>

a) Implement H .x/ in a Python function H\_eps(x, eps=0.01).

b) Make a Python function test\_H\_eps() for testing the implementation of

H\_eps. Check the values of some x < , x D , x D 0, x D , and some

x > .

Filename: smoothed\_Heaviside.140

3

Functions and Branching

Exercise 3.31: Implement an indicator function

In many applications there is need for an indicator function, which is 1 over some

interval and 0 elsewhere. More precisely, we define

(

I.xI L; R/ D

1; x 2 ŒL; R;

0; elsewhere

(3.27)

a) Make two Python implementations of such an indicator function, one with a di-

rect test if x 2 ŒL; R and one that expresses the indicator function in terms of

Heaviside functions (3.25):

I.xI L; R/ D H.x L/H.R x/ :

(3.28)

b) Make a test function for verifying the implementation of the functions in a).

Check that correct values are returned for some x < L, x D L, x D .L CR/=2,

x D R, and some x > R.

Filename: indicator\_func.

Exercise 3.32: Implement a piecewise constant function

Piecewise constant functions have a lot of important applications when modeling

physical phenomena by mathematics. A piecewise constant function can be defined

as

8

ˆ

v0 ; x 2 Œx0 ; x1 /;

ˆ

ˆ

ˆ

ˆ

ˆ

v

1 ; x 2 Œx1 ; x2 /;

ˆ

ˆ

ˆ

:

ˆ

< ::

(3.29)

f .x/ D

ˆ vi x 2 Œxi ; xi C1 /;

ˆ

ˆ

ˆ

ˆ

::

ˆ

ˆ

ˆ

ˆ :

:̂ v

x 2 Œxn ; xnC1

n

That is, we have a union of non-overlapping intervals covering the domain

Œx0 ; xnC1 , and f .x/ is constant in each interval. One example is the function

that is 1 on Œ0; 1, 0 on Œ1; 1:5, and 4 on Œ1:5; 2, where we with the notation in

(3.29) have x0 D 0; x1 D 1; x2 D 1:5; x3 D 2 and v0 D 1; v1 D 0; v3 D 4.

a) Make a Python function piecewise(x, data) for evaluating a piecewise con-

stant mathematical function as in (3.29) at the point x. The data object is a list

of pairs .vi ; xi / for i D 0; : : : ; n. For example, data is [(0, -1), (1, 0),

(1.5, 4)] in the example listed above. Since xnC1 is not a part of the data

object, we have no means for detecting whether x is to the right of the last in-

terval Œxn ; xnC1 , i.e., we must assume that the user of the piecewise function

sends in an x xnC1 .

b) Design suitable test cases for the function piecewise and implement them in

a test function test\_piecewise().

Filename: piecewise\_constant1.3.5 Exercises

141

Exercise 3.33: Apply indicator functions

Implement piecewise constant functions, as defined in Exercise 3.32, by observing

that

n

X

f .x/ D

vi I.xI xi ; xi C1 /;

(3.30)

i D0

where I.xI xi ; xi C1 / is the indicator function from Exercise 3.31.

Filename: piecewise\_constant2.

Exercise 3.34: Test your understanding of branching

Consider the following code:

def where1(x, y):

if x > 0:

print ’quadrant I or IV’

if y > 0:

print ’quadrant I or II’

def where2(x, y):

if x > 0:

print ’quadrant I or IV’

elif y > 0:

print ’quadrant II’

for x, y in (-1, 1), (1, 1):

where1(x,y)

where2(x,y)

What is printed?

Filename: simulate\_branching.

Exercise 3.35: Simulate nested loops by hand

Go through the code below by hand, statement by statement, and calculate the num-

bers that will be printed.

n = 3

for i in range(-1, n):

if i != 0:

print i

for i in range(1, 13, 2\*n):

for j in range(n):

print i, j

for i in range(1, n+1):

for j in range(i):

if j:

print i, j142

3

Functions and Branching

for i in range(1, 13, 2\*n):

for j in range(0, i, 2):

for k in range(2, j, 1):

b = i > j > k

if b:

print i, j, k

You may use a debugger, see Sect. F.1, or the Online Python Tutor10 , see Sect. 3.1.2,

to control what happens when you step through the code.

Filename: simulate\_nested\_loops.

Exercise 3.36: Rewrite a mathematical function

We consider the L.xI n/ sum as defined in Sect. 3.1.8 and the corresponding func-

tion L3(x, epsilon) function from Sect. 3.1.10. The sum L.xI n/ can be written

as

i

n

X

x

1

ci ; ci D

:

L.xI n/ D

i 1Cx

i D1

a) Derive a relation between ci and ci 1 ,

ci D aci 1 ;

where a is an expression involving i and x.

b) The relation ci D aci 1 means that we can start

P with term as c1 , and then in

each pass of the loop implementing the sum i ci we can compute the next

term ci in the sum as

term = a\*term

Write a new version of the L3 function, called L3\_ci(x, epsilon), that makes

use of this alternative computation of the terms in the sum.

c) Write a Python function test\_L3\_ci() that verifies the implementation of

L3\_ci by comparing with the original L3 function.

Filename: L3\_recursive.

Exercise 3.37: Make a table for approximations of cos x

The function cos.x/ can be approximated by the sum

C.xI n/ D

n

X

cj ;

j D0

where

cj D cj 1

and c0 D 1.

10

http://www.pythontutor.com/

x2

;

2j.2j 1/

j D 1; 2; : : : ; n;

(3.31)3.5 Exercises

143

a) Make a Python function for computing C.xI n/.

Hint Represent cj by a variable term, make updates term = -term\*... inside

a for loop, and accumulate the term variable in a variable for the sum.

b) Make a function for writing out a table of the errors in the approximation C.xI n/

of cos.x/ for some x and n values given as arguments to the function. Let the

x values run downward in the rows and the n values to the right in the columns.

For example, a table for x D 4; 6; 8; 10 and n D 5; 25; 50; 100; 200 can

look like

x

12.5664

18.8496

25.1327

31.4159

5

1.61e+04

1.22e+06

2.41e+07

2.36e+08

25

1.87e-11

2.28e-02

6.58e+04

6.52e+09

50

1.74e-12

7.12e-11

-4.87e-07

1.65e-04

100

1.74e-12

7.12e-11

-4.87e-07

1.65e-04

200

1.74e-12

7.12e-11

-4.87e-07

1.65e-04

Observe how the error increases with x and decreases with n.

Filename: cos\_sum.

Exercise 3.38: Use None in keyword arguments

Consider the functions L2(x, n) and L3(x, epsilon) from Sects. 3.1.8 and

3.1.10, whose program code is found in the file lnsum.py.

Make a more flexible function L4 where we can either specify a tolerance

epsilon or a number of terms n in the sum. Moreover, we can also choose

whether we want the sum to be returned or the sum and the number of terms:

value, n = L4(x, epsilon=1E-8, return\_n=True)

value = L4(x, n=100)

Hint The starting point for all this flexibility is to have some keyword arguments

initialized to an “undefined” value, called None, which can be recognized inside the

function:

def L3(x, n=None, epsilon=None, return\_n=False):

if n is not None:

...

if epsilon is not None:

...

One can also apply if n != None, but the is operator is most common.

Print error messages for incompatible values when n and epsilon are None or

both are given by the user.

Filename: L4.

Exercise 3.39: Write a sort function for a list of 4-tuples

Below is a list of the nearest stars and some of their properties. The list elements are

4-tuples containing the name of the star, the distance from the sun in light years, the144

3

Functions and Branching

apparent brightness, and the luminosity. The apparent brightness is how bright the

stars look in our sky compared to the brightness of Sirius A. The luminosity, or the

true brightness, is how bright the stars would look if all were at the same distance

compared to the Sun. The list data are found in the file stars.txt11, which looks

as follows:

data = [

(’Alpha Centauri A’,

4.3,

(’Alpha Centauri B’,

4.3,

(’Alpha Centauri C’,

4.2,

("Barnard’s Star",

6.0,

(’Wolf 359’,

7.7,

(’BD +36 degrees 2147’, 8.2,

(’Luyten 726-8 A’,

8.4,

(’Luyten 726-8 B’,

8.4,

(’Sirius A’,

8.6,

(’Sirius B’,

8.6,

(’Ross 154’,

9.4,

]

0.26,

0.077,

0.00001,

0.00004,

0.000001,

0.0003,

0.000003,

0.000002,

1.00,

0.001,

0.00002,

1.56),

0.45),

0.00006),

0.0005),

0.00002),

0.006),

0.00006),

0.00004),

23.6),

0.003),

0.0005),

The purpose of this exercise is to sort this list with respect to distance, apparent

brightness, and luminosity. Write a program that initializes the data list as above

and writes out three sorted tables: star name versus distance, star name versus ap-

parent brightness, and star name versus luminosity.

Hint To sort a list data, one can call sorted(data), as in

for item in sorted(data):

...

However, in the present case each element is a 4-tuple, and the default sorting of

such 4-tuples results in a list with the stars appearing in alphabetic order. This is not

what you want. Instead, we need to sort with respect to the 2nd, 3rd, or 4th element

of each 4-tuple. If such a tailored sort mechanism is necessary, we can provide our

own sort function as an argument to sorted. There are two alternative ways of

doing this.

A comparison function A sort user-provided sort function mysort(a, b) must

take two arguments a and b and return 1 if a should become before b in the sorted

sequence, 1 if b should become before a, and 0 if they are equal. In the present

case, a and b are 4-tuples, so we need to make the comparison between the right

elements in a and b. For example, to sort with respect to luminosity we can write

def mysort(a, b):

if a[3] < b[3]:

return -1

elif a[3] > b[3]:

return 1

else:

return 0

11

http://tinyurl.com/pwyasaa/funcif/stars.txt3.5 Exercises

145

The relevant call using this tailored sort function is

sorted(data, cmp=mysort)

A key function A quicker construction is to provide a key argument to sorted for

filtering out the relevant part of an object to be sorted. Here, we want to sort 4-

tuples, but use only one of the elements, say the one with index 3, for comparison.

Writing

sorted(data, key=lambda obj: obj[3])

will send all objects (4-tuples) through the key function whose return value is used

for the sorting. A lambda construction (see Sect. 3.1.14) is used to write the filtering

function inline.

Filename: sorted\_stars\_data.

Exercise 3.40: Find prime numbers

The Sieve of Eratosthenes is an algorithm for finding all prime numbers less than or

equal to a number N . Read about this algorithm on Wikipedia and implement it in

a Python program.

Filename: find\_primes.

Exercise 3.41: Find pairs of characters

Write a function count\_pairs(dna, pair) that returns the number of occur-

rences of a pair of characters (pair) in a DNA string (dna). For example, calling

the function with dna as ’ACTGCTATCCATT’ and pair as ’AT’ will return 2.

Filename: count\_pairs.

Exercise 3.42: Count substrings

This is an extension of Exercise 3.41: count how many times a certain string appears

in another string. For example, the function returns 3 when called with the DNA

string ’ACGTTACGGAACG’ and the substring ’ACG’.

Hint For each match of the first character of the substring in the main string, check

if the next n characters in the main string matches the substring, where n is the

length of the substring. Use slices like s[3:9] to pick out a substring of s.

Filename: count\_substr.

Exercise 3.43: Resolve a problem with a function

Consider the following interactive session:

>>> def f(x):

...

if 0 <= x <= 2:

...

return x\*\*2

...

elif 2 < x <= 4:

...

return 4

...

elif x < 0:

...

return 0

...146

3

Functions and Branching

>>> f(2)

4

>>> f(5)

>>> f(10)

Why do we not get any output when calling f(5) and f(10)?

Hint Save the f value in a variable r and do print r.

Filename: fix\_branching.

Exercise 3.44: Determine the types of some objects

Consider the following calls to the makelist function from Sect. 3.1.6:

l1 = makelist(0, 100, 1)

l2 = makelist(0, 100, 1.0)

l3 = makelist(-1, 1, 0.1)

l4 = makelist(10, 20, 20)

l5 = makelist([1,2], [3,4], [5])

l6 = makelist((1,-1,1), (’myfile.dat’, ’yourfile.dat’))

l7 = makelist(’myfile.dat’, ’yourfile.dat’, ’herfile.dat’)

Simulate each call by hand to determine what type of objects that become elements

in the returned list and what the contents of value is after one pass in the loop.

Hint Note that some of the calls will lead to infinite loops if you really perform the

above makelist calls on a computer.

You can go to the Online Python Tutor12 , paste in the makelist function and

the session above, and step through the program to see what actually happens.

Filename: find\_object\_type.

Remarks This exercise demonstrates that we can write a function and have in mind

certain types of arguments, here typically int and float objects. However, the

function can be used with other (originally unintended) arguments, such as lists and

strings in the present case, leading to strange and irrelevant behavior (the problem

here lies in the boolean expression value <= stop which is meaningless for some

of the arguments).

Exercise 3.45: Find an error in a program

For the formula

f .x/ D e rx sin.mx/ C e sx sin.nx/

we have made the program

def f(x, m, n, r, s):

return expsin(x, r, m) + expsin(x, s, n)

x = 2.5

print f(x, 0.1, 0.2, 1, 1)

12

http://www.pythontutor.com/3.5 Exercises

147

from math import exp, sin

def expsin(x, p, q):

return exp(p\*x)\*sin(q\*x)

Running this code results in

NameError: global name ’expsin’ is not defined

What is the problem? Simulate the program flow by hand, use the debugger to step

from line to line, or use the Online Python Tutor. Correct the program.

Filename: find\_error\_undef.4

User Input and Error Han