Computational programming python

# Getting Started

A computer does **two things,** and two things only: it performs **calculations** and it **remembers the results** of those calculations. But it does those two things extremely well.

Even with the speed of modern computers, there are still problems that are beyond modern computational models (e.g., understanding climate change)

**Computational Thinking**

Knowledge is either **Declarative** or **imperative.**

**Declarative knowledge** is composed of **statements of fact.** For example, “the square root of x is a number y such that y\*y =x “ 🡪 It doesn’t tell us how to find square root.

**Imperative knowledge** is **“how to”** knowledge, or **recipes** for deducing information.

**Finding the square root**

1. Start with a guess, g.

2. If g\*g is close enough to x, stop and say that g is the answer.

3. Otherwise create a new guess by averaging g and x/g, i.e., (g +x/g)/2.

4. Using this new guess, which we again call g, repeat the process until g\*g is close enough to x.

An **algorithm** is a finite list of instructions that describe a computation that when executed on a set of inputs will proceed through a set of well-defined states and eventually produce an output. An algorithm is a bit like recipe

Fixed program computers and Stored program computers

In some cases, it performs a test, and on the basis of that test, execution may jump to some other point in the sequence of instructions. This is called **flow of control**, and is essential to allowing us to write programs that perform complex tasks.

The **syntax** describes which strings of words constitute well-formed sentences, the **static semantics** defines which sentences are meaningful, and the **semantics** defines the meaning of those sentences.

If a program has no syntactic errors and no static semantic errors it has a meaning, that is it has semantics.

When programs is not written correctly or means something then it might crash or it might keep running and running and never stop. Or it might run to completion and produce wrong answers

# Introduction to python

Through each programming language is different, there are some dimensions along which they can be related.

**Low level vs High level** refers to whether we program using instructions and data objects at the level of the machine or whether we program using more abstract operations that have been provided by the language designers.

**General vs Targeted to an application domain** refers to whether the primitive operations of the programming language are widely applicable or are fine tuned to a domain.

**Interpreted vs Compiled** refers to whether the sequence of instructions written by the programmer, called source code, is executed directly (by an interpreter) or whether it is first converted (by a compiler) into a sequence of machine-level primitive operations**.**

**--------------------------------------------------------------------------------------------------------------------------**

It is often easier to debug programs written in languages that are designed to be interpreted, because the interpreter can produce error messages that are easy to correlate with the source code.

Compiled languages usually produce programs that run more quickly and use less space.

The Basic Elements of Python

A Python **program**, sometimes called a **script,** is a sequence of definitions and commands. These definitions are evaluated and the commands are executed by the Python interpreter in something called the **shell.**

A **command,** often called a **statement,** instructs the interpreter to do something.

For example, the statement **print('Yankees rule!')** instructs the interpreter to call the function **print,** which will output the string Yankees rule! to the window associated with the shell.

**Objects, Expressions, and Numerical Types**

**Objects** are the core things that Python programs manipulate. Every object has a **type** that defines **the kinds of things** that programs can do with that object.

**Types are either scalar or non-scalar.**

* **Scalar** objects are **indivisible.** Think of them as the atoms of the language.
* **Non-scalar** objects, for example strings, **have internal structure.**

Many types of objects can be denoted by **literals** in the text of a **program**. For example,

the text 2 is a literal representing a number and the text 'abc' a literal representing a string.

**Python has four types of scalar objects:**

* **int** is used to represent integers. Literals of type int are 4 -8 16 1000
* **float** is used to represent real numbers. Literals of type float always include a decimal point (e.g., 3.0 or 3.17 or -28.72).
* **bool** is used to represent the Boolean values **True** and **False.**
* **None** is a type with a single value.

Objects and **operators** can be combined to form **expressions,** each of which evaluates to an object of some type. We will refer to this as **the value** of the expression.

**For example,** the expression 3 + 2 denotes the object 5 of type int

The built-in Python function **type** can be used to find out the type of an object:

type(3) 🡪 type ‘int’

type(3.0) 🡪 type ‘float’

Operators on objects of type int and float are Comparison Operators

i + j == Equal

i – j != Not Equal

i \* j > Greater than

i // j >= Greater than or equal

i / j < Lesser than

i % j <= Lesser than or equal

i \*\* j

The arithmetic operators have the usual precedence **PODMAS**

The primitive operators on type bool are **and, or, and not**

**Variables and Assignment**

**Variables** provide a way to associate **names with objects.** Consider the code

pi = 3

radius = 11

area = pi \* (radius\*\*2)

radius = 14

It first **binds** the names **pi** and **radius** to different objects of **type int.** It then **binds** the

**name area** to a third object of **type int.**

If the program then executes radius = 14, the name radius is rebound to a different object of **type int** Note that this assignment has no effect on the value to which area is bound. It is still bound to the object denoted by the expression 3\*(11\*\*2).

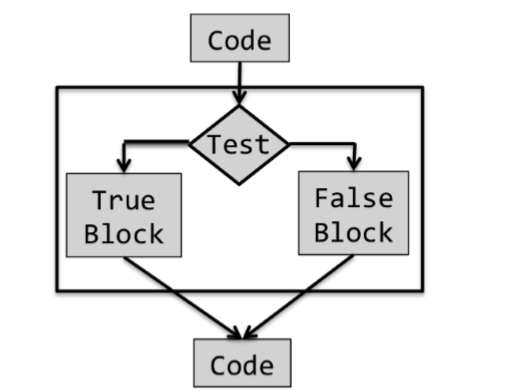
Branching Programs

The kinds of computations we have been looking at thus far are called **straight line programs.** They execute one statement after another in the order in which they appear, and stop when they run out of statements.

**Branching** programs are more interesting. The simplest branching statement is a **conditional.** A Conditional statement has three parts

* **a test,** i.e., an expression that evaluates to either **True or False;**
* **a block of code** that is executed if the test evaluates to **True;** and
* **an optional block of code** that is executed if the test evaluates to **False.**

After a conditional statement is executed, execution resumes at the code following the statement.



In Python, a conditional statement has the form

if Boolean expression:

block of code

else:

block of code

When the statement is true the first block of code is executed and when it is false the else block of code is executed

We use **Indentation** to represent the block of the codes

When either the true block or the false block of a conditional contains another conditional, the conditional statements are said to be **nested.**

if x%2 == 0:

if x%3 == 0:

print('Divisible by 2 and 3')

else:

print('Divisible by 2 and not by 3')

elif x%3 == 0:

print('Divisible by 3 and not by 2')

Strings and Input

Objects of type str are used to represent strings of characters. Literals of type str can be written using either single or double quotes, e.g., 'abc' or "abc".

The **operator +** is said to be **overloaded:** It has different meanings depending upon the types of the objects to which it is applied.

4 + 4 🡪 8

‘a’ + ‘a’ 🡪 ‘aa’

‘a’ \* ‘a’ 🡪 Error

That **type checking** exists is a good thing. It turns careless (and sometimes subtle) mistakes into errors that stop execution, rather than errors that lead programs to behave in mysterious ways.

Strings are one of several sequence types in Python. They share the following operations with all sequence types 🡪 **Length len()** , **Indexing** and **Slicing**

**Input**

Python 3 has a function, **input,** that can be used to get input directly from a user. It takes a string as an argument and displays it as a prompt in the shell.

The line typed by the user is treated as a **string** and becomes **the value** returned by the function.

name = input('Enter your name: ')

n = input('Enter an int: ')

When you enter a number to the input it is bound to a string and not int

**Type Conversion**

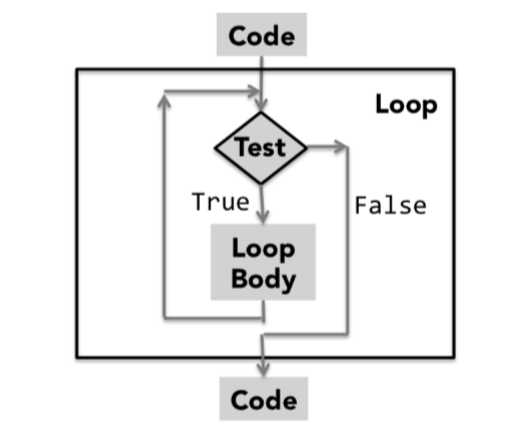
whenever a string is a valid literal of some type, a **type conversion** can be applied to it.

Type conversions (also called type casts) are used often in Python code. We use the name of a type to convert values to that type.

**int(‘4’) 🡪 This converts ‘4’ of string object to object of type int 4**

Iteration

Like a conditional statement, it begins with a test. If the test evaluates to True, the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement.



# Square an integer, the hard way

x = 3

ans = 0

itersLeft = x

while (itersLeft != 0):

ans = ans + x

itersLeft = itersLeft - 1

print(str(x) + '\*' + str(x) + ' = ' + str(ans))

The Condition is False during fourth test then the flow of control proceeds to the next statement

It is sometimes convenient to exit a loop without testing the loop condition. Executing a **break statement** terminates the loop in which it is contained, and transfers control to the code immediately following the loop.

#Find a positive integer that is divisible by both 11 and 12

x = 1

while True:

if x%11 == 0 and x%12 == 0:

break

x = x + 1

print(x, 'is divisible by 11 and 12')

If a break statement is executed inside a **nested loop** (a loop inside another loop), the break will terminate the **inner loop.**

# Some Simple Numerical Programs

Exhaustive Enumeration

#Find the cube root of a perfect cube

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

ans = 0

while ans\*\*3 < abs(x):

ans = ans + 1

if ans\*\*3 != abs(x):

print(x, 'is not a perfect cube')

else: if x < 0:

ans = -ans

print('Cube root of', x,'is', ans)

Whenever you write **a loop,** you should think about an appropriate **decrementing function.**

The algorithmic technique used in this program is a **variant of guess and check** called **exhaustive enumeration.** We enumerate all possibilities until we get to the right answer or **exhaust the space of possibilities.**

**exhaustive enumeration algorithms** are often the most practical way to solve a problem. They are typically **easy to implement and easy to understand.** And, in many cases, they run fast enough for all practical purposes.

For loops

The while loops we have used so far are highly stylized.

Each iterates over a sequence of integers. Python provides a language mechanism, the for loop, that can be used to simplify programs containing this kind of iteration.

**for variable in sequence:**

**code block**

The variable following for is bound to the first value in the sequence, and the code block is executed. The variable is then assigned the second value in the sequence, and the code block is executed again. The process continues until the sequence is exhausted or a break statement is executed within the code block.

The sequence of values bound to variable is most commonly generated using the built-in function range, which returns a series of integers. The range function takes three integer arguments: start, stop, and step.

The numbers in the progression are generated on an **“as needed” basis,** so even expressions such as **range(1000000)** consume little memory

x = 4

for i in range(0, x): #The x value inside the loop doesn’t affect the number of iterations

print(i) The value of x is executed before entering the loop and the line

x = 5 command is not executed again that is only the inside of the loops

executed till the iteration ends

x = 4

for j in range(x): # This prints out: 0 1 2 3 0 1 0 1 0 1

for i in range(x):

print(i)

x = 2

The range function in the outer loop is evaluated only once, but the range function in the inner loop is evaluated each time the inner for statement is reached.

#Find the cube root of a perfect

cube x = int(input('Enter an integer: '))

for ans in range(0, abs(x)+1):

if ans\*\*3 >= abs(x):

break

if ans\*\*3 != abs(x):

print(x, 'is not a perfect cube')

else: if x < 0:

ans = -ans print('Cube root of', x,'is', ans)

The for statement can be used in conjunction with the in operator to conveniently iterate over characters of a string and other sequence type objects.

Approximate solutions and Bisection Search

what should the program do if asked to find the square root of 2?

The square root of 2 is not a rational number. This means that there is no way to precisely represent its value as a finite string of digits (or as a float), so the problem as initially stated cannot be solved.

The right thing to have asked for is a program that finds an approximation to the square root—i.e., an answer that is close enough to the actual square root to be useful. let’s think of “close enough” as an answer that lies within some constant, call it epsilon, of the actual answer.

x = 25

epsilon = 0.01

step = epsilon\*\*2

numGuesses = 0

ans = 0.0

while abs(ans\*\*2 - x) >= epsilon and ans <= x:

ans += step

numGuesses += 1

print('numGuesses =', numGuesses)

if abs(ans\*\*2 - x) >= epsilon:

print('Failed on square root of', x)

else:

print(ans, 'is close to square root of', x)

When the code is run, it prints numGuesses = 49990

4.999000000001688 is close to square root of 25

What do you think will happen if we set x = 0.25? Will it find a root close to 0.5 ? Nope.

It will report numGuesses = 2501 Failed on square root of 0.25

Exhaustive enumeration is a search technique that works only if the set of values being searched includes the answer. In this case, we are enumerating the values between 0 and the value of x. When x is between 0 and 1, the square root of x does not lie in this interval. One way to fix this is to change the second operand of and in the first line of the while loop to get while abs(ans\*\*2 - x) >= epsilon and ans\*ans <= x:

The program will execute the while loop at most x/step times. Let’s try the code on something bigger, e.g., x = 123456. It will run for a bit, and then

print numGuesses = 3513631 Failed on square root of 123456

Surely there exists a floating point number that approximates the square root of 123456 to within 0.01. Why didn’t our program find it? The problem is that our step size was too large, and the program skipped over all the suitable answers.

Try making step equal to epsilon\*\*3 and running the program. It will eventually find a suitable answer, but you might not have the patience to wait for it to do so.

350 million guesses to find a satisfactory answer.

Bisection Search

Start with 0----------------max and start searching that interval. Since we don’t necessarily know where to start searching, let’s start in the middle.

0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_guess\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_max If that is not the right answer (and it won’t be most of the time), ask whether it is too big or too small. If it is too big, we know that the answer must lie to the left. If it is too small, we know that the answer must lie to the right.

x = 25

epsilon = 0.01

numGuesses = 0

low = 0.0

high = max(1.0, x)

ans = (high + low)/2.0

while abs(ans\*\*2 - x) >= epsilon:

print('low =', low, 'high =', high, 'ans =', ans)

numGuesses += 1

if ans\*\*2 < x:

low = ans

else:

high = ans

ans = (high + low)/2.0

print('numGuesses =', numGuesses) print(ans, 'is close to square root of', x)

numGuesses = 13

5.00030517578125 is close to square root of 25

Notice that at each iteration of the loop the size of the space to be searched is cut in half. Because it divides the search space in half at each step, it is called a bisection search. Bisection search is a huge improvement over our earlier algorithm, which reduced the search space by only a small amount at each iteration.

A Few words about using floats

for i in range(10):

x = x + 0.1 This gives 0.9999999999999999 is not 1.0

This is due to the fact the floats are represented by binary numbers and not in base 10 numbers

0.625 (5/8) is represented as the pair ( 101, -11) because 5/8 is 0.101 in binary and -11 is the binary representation of -3, the pair stands for 5\*2^-3 = 5/8 = 0.625

The decimal fraction 1/10, which we write in python as 0.1

The best we can do with four significant binary digits is ( 0011, -101) This is equivalent to 3/32 that is 0.09375

If we had five significant binary digits, we would represent 0.1 as ( 11001, -1000) which is equivalent to 25/156 that is 0.09765625

To get exact 1.0 we need infinite number of digits

In most Python implementations, there are 53 bits of precision available for floating point numbers

The expression **round(x, numDigits)** returns the floating point number equivalent to rounding the value of x to numDigits decimal digits following the decimal point.

print round(2\*\*0.5, 3) 🡪 will print 1.414

When comparing floating points numbers it is better to use abs(x-y) < 0.0001 than x == y

Newton-Raphson

It can be used to find the real roots of many functions

Newton proved a theorem that implies that if a value, call it guess, is an approximation to a root of a polynomial, then guess – p(guess)/p’(guess), where p’ is the first derivative of p, is a better approximation.

We know that we can improve on the current guess, call it y, by choosing as our next guess y – (y^2 – k)/2y, This is called Successive approximation.

#Newton-Raphson for square root

#Find x such that x\*\*2 - 24 is within epsilon of 0

epsilon = 0.01

k = 24.0

guess = k/2.0

while abs(guess\*guess - k) >= epsilon:

guess = guess - (((guess\*\*2) - k)/(2\*guess))

print('Square root of', k, 'is about', guess)

# Functions, Scoping and Abstraction

The above program is a reasonable piece of code, but it **lacks general utility.** It works only for values denoted by the variables x and epsilon. This means that if we want to reuse it, we need to copy the code

Because of this we cannot easily use this computation inside of some other, more complex, computation.

Furthermore, if we want to compute cube roots rather than square roots, we have to edit the code. If we want a program that computes both square and cube roots (or for that matter square roots in two different places), the program would contain multiple chunks of almost identical code. This is a very bad thing.

And Functions make it **easier to fix and debug the codes,** instead of debugging same problems in the program multiple times

Function Definitions

In Python each function definition is of the form

**def** **name of function**(list of formal parameters):

**body of function**

def **maxVal(x, y):**

if x > y:

return x

else:

return y

**def** is a reserved word that tells Python that a function is about to be **defined.** The function name (maxVal in this example) is simply a name that is used to refer to the function.

The sequence of **names within the parentheses** following the function name (x,y in this example) are **the formal parameters of the function.** When the function is used, the formal parameters are bound (as in an assignment statement) to the actual parameters (often referred to as arguments) of the function invocation (also referred to as a function call). For example, the invocation

maxval(3, 4) 🡪 x binds to 3 and y binds to 4

A special statement, **return,** that can be used only within the body of a function.

**A function call is an expression**, and like all expressions it has a value. That value is the value returned by the invoked function.

maxVal(3,4)\*maxVal(3,2) is 12

Note that **execution of a return** statement **terminates** an invocation of **the function** and any code after it is not executed

If there **is no return statement** or return value the function returns or **gives None datatype**

Parameters provide something called lambda abstraction, allowing programmers to write code that manipulates not specific objects, but instead whatever objects the caller of the function chooses to use as actual parameters.

Keyword Arguments and Default Values

In Python, there **are two ways** that **formal parameters get bound to actual parameters.**

**positional**—the first formal parameter is bound to the first actual parameter, the second formal to the second actual

**keyword arguments,** in which formals are bound to actuals using the name of the formal parameter.

def printName(firstName, lastName, reverse):

if reverse:

print(lastName + ', ' + firstName)

else:

print(firstName, lastName)

printName('Olga', 'Puchmajerova', False)

printName('Olga', 'Puchmajerova', reverse = False)

printName('Olga', lastName = 'Puchmajerova', reverse = False)

printName(lastName = 'Puchmajerova', firstName = ' Olga', reverse = False)

Though the keyword arguments can appear in any order in the list of actual parameters, **it is not legal to follow a keyword argument with a non-keyword argument.** Therefore, an error message would be produced by

printName('Olga', lastName = 'Puchmajerova', False)

Keyword arguments are commonly used in conjunction with **default parameter values.**

def printName(firstName, lastName, reverse = False):

Default values allow programmers to call a function with fewer than the specified number of arguments.

printName('Olga', 'Puchmajerova')

printName('Olga', 'Puchmajerova', True)

printName('Olga', 'Puchmajerova', reverse = True)

will print

Olga Puchmajerova

Puchmajerova, Olga

Puchmajerova, Olga

Scoping

Let’s look at another small example

def f(x): #name x used as formal parameter

y = 1

x = x + y

print('x =', x)

return x

x = 3

y = 2

z = f(x) #value of x used as actual parameter

print('z =', z)

print('x =', x)

print('y =', y)

When run, this code prints,

x = 4

z = 4

x = 3

y = 2

**At the call of f,** the formal parameter x is locally bound to the value of the actual parameter x. It is important to note that though the actual and formal parameters have the same name, they are not the same variable. Each function defines a new name space, also called a **scope.** The **formal parameter x** and the **local variable y** that are used in f **exist only within the scope of the definition of f.** The assignment statement x = x + y within the function body binds the local name x to the object. The **assignments in f** **have no effect** at all on the **bindings of the names x and y** that **exist outside the scope of f.**

1. At top level, i.e., the level of the shell, a symbol table keeps track of all names defined at that level and their current bindings.

2. When a function is called, a new symbol table (often called **a stack frame**) is created. This table keeps track of all names defined within the function (including the formal parameters) and their current bindings. If a function is called from within the function body, yet another stack frame is created.

3. When the **function completes, its stack frame goes away.**

def f():

print(x)

def g(): #This produces **UnboundLocalError:** local variable 'x' referenced

print(x) before assignment

x = 1

This happens because the assignment statement following the print statement causes x to be local to g. And because x is local to g, it has no value when the print statement is executed.

Specifications

Defines a function, findRoot, that generalizes the bisection search we used to find square roots. It also contains a function, testFindRoot, that can be used to test whether or not findRoot works as intended.

def findRoot(x, power, epsilon):

"""Assumes x and epsilon int or float, power an int,

epsilon > 0 & power >= 1

Returns float y such that y\*\*power is within epsilon of x.

If such a float does not exist, it returns None"""

if x < 0 and power%2 == 0: #Negative number has no even-powered roots

return None

low = min(-1.0, x)

high = max(1.0, x)

ans = (high + low)/2.0

while abs(ans\*\*power - x) >= epsilon:

if ans\*\*power < x:

low = ans

else:

high = ans

ans = (high + low)/2.0

return ans

def testFindRoot():

epsilon = 0.0001

for x in [0.25, -0.25, 2, -2, 8, -8]:

for power in range(1, 4):

print('Testing x =', str(x), 'and power = ', power)

result = findRoot(x, power, epsilon)

if result == None:

print(' No root')

else:

print(' ', result\*\*power, '~=', x)

To inexperienced programmers, writing test functions such as this often seems to be a waste of effort. Experienced programmers know, however, that an investment in writing testing code often pays big dividends. It certainly beats typing test cases into the shell over and over again during debugging (the process of finding out why a program does not work, and then fixing it). It also forces us to think about which tests are likely to be most illuminating.

The text between the triple quotation marks is called a **docstring in Python.** By convention, Python programmers use **docstrings to provide specifications of functions.** These docstrings can be accessed using the built-in function help.

**help(abs)**

A **specification** of a function defines a contract between the implementer of a function and those who will be writing programs that use the function clients

**This Contracts contains two parts**

* **Assumptions:** These describe conditions that must be met by **clients of the function.** Typically, they describe constraints on the actual parameters. Almost always, **they specify the acceptable set of types for each parameter**
* **Guarantees:** These describe conditions that must be met by the function, **provided** that it has been called in a way that satisfies the assumptions.

**Functions** are a way of **creating computational elements** that we can think of **as primitives.** Just as we have the built-in functions **max** and **abs,** we would like to have the equivalent of a built-in function for **finding roots** and for **many other complex operations.**

**Functions facilitate** this by providing **decomposition and abstraction.**

* **Decomposition** creates structure. It allows us to **break a program into parts** that are **reasonably self-contained,** and that may be reused in different settings.
* **Abstraction** hides detail. It allows us to use a piece of code as if it were **a black box**—that is, something whose interior details we cannot see, don’t need to see, and shouldn’t even want to see.

The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

This is the way organizations go about using teams of programmers to get things done. Given a specification of a module, a programmer can work on implementing that module without worrying unduly about what the other programmers on the team are doing. Moreover, the other programmers can use the specification to start writing code that uses that module without worrying unduly about how that module is to be implemented.

The **specification** of **findRoot** is an **abstraction** of all the possible implementations that meet the specification. Clients of findRoot can assume that the implementation meets the specification, but they should assume nothing more.

Recursion

In general, a recursive definition is made up of **two parts.** There is **at least one base case** that directly specifies the result for a special case and there is **at least one recursive (inductive) case** that defines the answer in terms of the answer to the question on some other input, typically a simpler version of the same problem.

def factR(n):

“”” Assumes n an int > 0

Returns n! “””

If n == 1:

return n

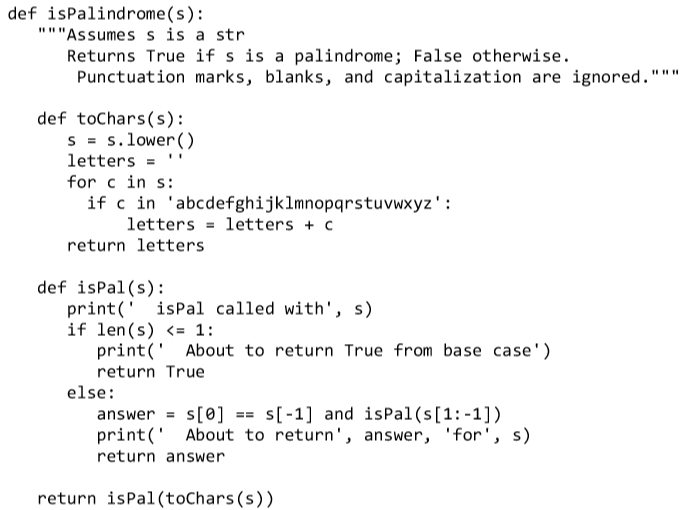
else:

return n\*factR(n-1)

When function make a recursive call, it always does so with a value one less than the value with which it was called. Eventually, the recursion terminates with the call factR(1).

**Palindromes**

The function isPalindrome contains two internal helper functions.



Global Variables

In a function, the line of code global numFibcalls tells python that the name numFibcalls should be defined at the outermost scope of the module in which the line of code appears rather than within the scope of the function in which the line of code appears.

Had we not included the code global numFibCalls, the name numFibCalls would have been local to each of the functions fib and testFib

def fib(x):

"""Assumes x an int >= 0

Returns Fibonacci of x"""

global numFibCalls

numFibCalls += 1

if x == 0 or x == 1:

return 1

else:

return fib(x-1) + fib(x-2)

def testFib(n):

for i in range(n+1):

global numFibCalls

numFibCalls = 0

print('fib of', i, '=', fib(i))

print('fib called', numFibCalls, 'times.')

The key to making programs readable is **locality.** One reads a program a piece at a time, and the less context needed to understand each piece, the better. Since global variables can be modified or read in a wide variety of places, the sloppy use of them can destroy locality. Nevertheless, there are times when they are just what is needed.

Modules

So far, we have operated under the assumption that our entire program is stored in one file. This is perfectly reasonable as long as programs are small. As programs get larger, however, it is typically more convenient to store different parts of them in different files. Imagine, for example, that multiple people are working on the same program. It would be a nightmare if they were all trying to update the same file. Python modules allow us to easily construct a program from code in multiple files.

A module is a .py file containing Python definitions and statements. We could create, for example, a file circle.py containing the code

pi = 3.14159

def area(radius):

return pi\*(radius\*\*2)

def circumference(radius):

return 2\*pi\*radius

def sphereSurface(radius):

return 4.0\*area(radius)

def sphereVolume(radius):

return (4.0/3.0)\*pi\*(radius\*\*3)

A program gets access to a module through an import statement.

import circle

pi = 3

print(pi)

print(circle.pi)

print(circle.area(3))

3

3.14159

28.27431

Executing import M creates a binding for module M in the scope in which the import appears. Therefore, in the importing context we use dot notation to indicate that we are referring to a name defined in the imported module. For example, outside of circle.py, the references pi and circle.pi can refer to different objects.

When one imports a module one often has no idea what local names might have been used in the implementation of that module. The use of dot notation to fully qualify names avoids the possibility of getting burned by an accidental name clash.

For example, executing the assignment pi = 3 outside of the circle module does not change the value of pi used within the circle module.

There is a variant of the import statement that allows the importing program to omit the module name when accessing names defined inside the imported module.

Executing the statement from M import \* creates bindings in the current scope to all objects defined within M, but not to M itself.

from circle import \*

print(pi) 🡪 Prints 3.14159

print(circle.pi) 🡪 NameError: name ‘circle’ is not defined

# STRUCTURED TYPES, MUTABILITY, AND HIGHER-ORDER FUNCTIONS

The numeric types int and float are scalar types. That is to say, objects of these types have no accessible internal structure. In contrast, str can be thought of as a structured, or non-scalar, type.

Lets look at other data types in python such as tuples, list, range and dictionaries

Tuples

Like strings, tuples are immutable **ordered sequences** of elements. The individual elements can be of **any type,** and need not be of the same type as each other. Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.

**T1 = ()**

**T2 = (1, ‘two’, 3)**

For single element tuple include comma 🡪 **(2,)**

Like strings, **tuples** can be **concatenated, indexed, and sliced.**

**Sequences and Multiple Assignment**

If you know the length of a sequence (e.g., a tuple or a string), it can be convenient to use Python’s **multiple assignment** statement to extract the individual elements.

**X, Y = (3, 4)**

X will be bound to 3 and Y to 4

def **findExtremeDivisors(n1, n2):**

"""Assumes that n1 and n2 are positive ints

Returns a tuple containing the smallest common divisor > 1 and

the largest common divisor of n1 and n2. If no common divisor,

returns (None, None)"""

minVal, maxVal = None, None

for i in range(2, min(n1, n2) + 1):

if n1%i == 0 and n2%i == 0:

if minVal == None:

minVal = i

maxVal = i

return (minVal, maxVal)

minDivisor, maxDivisor = findExtremeDivisors(100, 200)

will bind minDivisor to 2 and maxDivisor to 100.

Ranges

Like strings and tuples, ranges are immutable. The range function **returns an object of type range.** The range function takes three integer arguments: start, stop, and step, and returns the progression of integers start, start + step, start + 2\*step

If only two arguments are supplied, a step of 1 is used. If only one argument is supplied, that argument is the stop, start defaults to 0, and step defaults to 1.

All of the operations on tuples are also available for ranges, except for concatenation and repetition.

Unlike objects of type tuple, the amount of space **occupied** by an object of **type range**

is not proportional to its length. Because a range is fully defined by its start, stop, and step values; it can be stored in a small amount of space.

Lists and Mutability

A list is an ordered sequence of values, where each value is **identified by an index.**

L = [ ] 🡪 Empty list

Like strings and tuples, **lists** can **be concatenated, indexed and sliced**

Unlike tuples, **lists are mutable** that is the list can be modified or altered

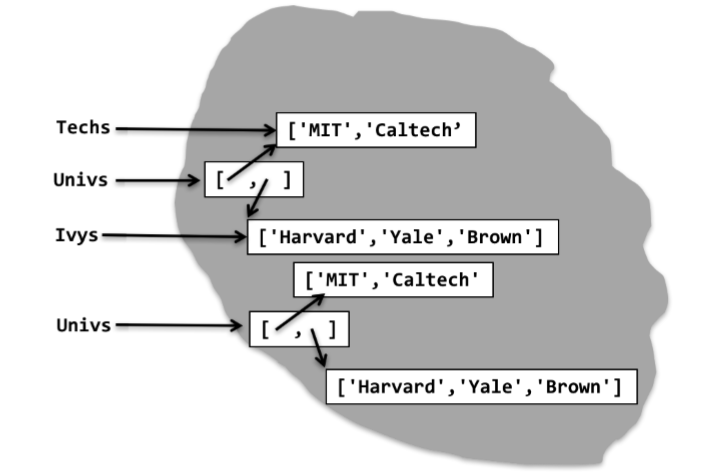
Techs = ['MIT', 'Caltech']

Ivys = ['Harvard', 'Yale', 'Brown']

Univs = [Techs, Ivys]

Univs1 = [ ['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown'] ]

Univs == Univs1 🡪 True



The Univs and Univs1 are bound to different objects can be verified using built in function **id,** which returns a unique integer identifier for an object.

id(Univs), id(Univs1)

The elements of Univs are not copies of the lists to which Techs and Ivys are bound, but are rather the lists themselves.

**Techs.append('RPI')**

The **append** method has a side effect. Rather than create a new list, **it mutates the existing list** Techs by adding a new element, the string 'RPI', to the end of it.

This doesn’t just mutate **Techs** it also modifies **Univs** since they are bound to same objects

**Aliasing** - There are two distinct paths to the same list object. One path is through the **variable Techs** and the other through the **first element** of the list object to which **Univs** is bound. One can mutate the object via either path, and the effect of the mutation will be visible through both paths.

**Techs.append(Ivys)** 🡪 The result is a list contains a list

Instead if you want to add the elements of one list into another use **list concatenation** or the **extend** method

Some of the method associated with lists, Note that all of these **except count** and **index mutate the list.**

**L.append(e)** adds the object e to the end of L.

**L.count(e)** returns the number of times that e occurs in L.

**L.insert(i, e)** inserts the object e into L at index i.

**L.extend(L1)** adds the items in list L1 to the end of L.

**L.remove(e)** deletes the first occurrence of e from L.

**L.index(e)** returns the index of the first occurrence of e in L, raises an exception if e is not in L.

**L.pop(i)** removes and returns the item at index i in L, raises an exception if L is empty. If i is omitted, it defaults to -1, to remove and return the last element of L.

**L.sort()** sorts the elements of L in ascending order.

**L.reverse()** reverses the order of the elements in L.

Def removeDups(L1, L2):

for e1 in L1:

if e1 in L2:

L1.remove(e1)

This function has unnecessary side effects due to mutating of the list

Slicing is not the only way to clone lists in Python. The expression **list(L)** returns a copy of the list L. If the list to be copied contains mutable objects that you want to copy as well, **import** the standard library **module copy** and use the function **copy.deepcopy.**

List Comprehension

List comprehension provides a concise way to apply an operation to the values in a sequence. It creates a new list in which each element is the result of applying a given operation to a value from a sequence

**L = [x\*\*2 for x in range(1,7)]**

The for clause in a list comprehension can be followed by one or more if and for statements that are applied to the values produced by the for clause. These additional clauses modify the sequence of values generated by the first for clause and produce a new sequence of values, to which the operation associated with the comprehension is applied.

mixed = [1, 2, 'a', 3, 4.0]

**print([x\*\*2 for x in mixed if type(x) == int])**

Prints [1, 4, 9]

Functions as Objects

In Python, functions are first-class objects. That means that they can be **treated like objects of any other type**

Functions have types, they can appear in expressions, e.g., as the right-hand side of an assignment statement or as an argument to a function; they can be elements of lists

Using functions as arguments allows a style of coding called **higher-order programming.** It can be particularly convenient in conjunction with lists

def **applyToEach(L, f):**

"""Assumes L is a list, f a function

Mutates L by replacing each element, e, of L by f(e)"""

for i in range(len(L)):

L[i] = f(L[i])

applyToEach(L, abs)

applyToEach(L, int)

applyToEach(L, factR)

applyToEach(L, fib)

The **function applyToEach** is called **higher-order** because it has an **argument that is itself a function.**

Python has a built-in higher-order function, **map,** that is similar to, but more general than, the applyToEach function, it is designed to be used in conjunction with a for loop. In its simplest form the first argument to map is a unary function (i.e., a function that has only one parameter) and the second argument is any ordered collection of values suitable as arguments to the first argument.

When used in a for loop, **map behaves like the range function** in that it returns one value for each iteration of the loop

**for i in map(fib, [2, 6, 4]):**

print(i)

More generally, the first argument to map can be a function of n arguments, in which case it must be followed by n subsequent ordered collections

**for i in map(min, L1, L2):**

print(i)

Python supports the creation of **anonymous functions** using the reserved word **lambda.** The general form of a **lambda expression** is

**lambda <sequence of variable names>: <expression>**

Lambda expressions are frequently used as arguments to higher-order functions.

L = []

**for i in map(lambda x, y: x\*\*y, [1 ,2 ,3, 4], [3, 2, 1, 0]):**

L.append(i)

print(L)

prints [1, 4, 3, 1]

Strings

Methods associated with strings

**s.count(s1)** counts how many times the string s1 occurs in s.

**s.find(s1)** returns the index of the first occurrence of the substring s1 in s, and -1 if **s1 is not in s.**

**s.rfind(s1)** same as find, but starts from the end of s (the “r” in rfind stands for reverse).

**s.index(s1)** same as find, but raises an exception (Chapter 7) if s1 is not in s.

**s.rindex(s1)** same as index, but starts from the end of s.

**s.lower()** converts all uppercase letters in s to lowercase.

**s.replace(old, new)** replaces all occurrences of the string old in s with the string new. **s.rstrip()** removes trailing white space from s.

**s.split(d)** Splits s using d as a delimiter.

Dictionaries

Objects of type dict are like lists except that we **index them using keys.** Think of a dictionary as **a set of key/value pairs.** Dictionaries are mutable like lists

**monthNumbers** = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5, 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}

The entries in a dict are **unordered** and **cannot be accessed** with an **index.**

Like lists, dictionaries are mutable. We can add an entry by writing

**monthNumbers[‘June’] = 6**

or change an entry

**monthNumbers['May'] = 'V'**

A for statement can be used to iterate over the entries in a dictionary. However, the value assigned to **the iteration variable is a key,** not a key/value pair. **The order** in which the keys are seen in the iteration is **not defined.**

The **method keys returns** an object of type dict\_keys. This is an example of a view object.

Objects of type dict\_keys can be iterated over using for, and membership can be tested using in. An object of type dict\_keys can easily be converted into a list, **using list(dict.keys())**

**Not all types** of objects can **be used as keys:** **A key must be** an object of a **hashable type.**

A type is hashable if it has

• A **\_\_hash\_\_** method that maps an object of the type to an int, and for every object the value returned by \_\_hash\_\_ does not change during the lifetime of the object, and

• An **\_\_eq\_\_** method that is used to compare objects for equality.

All of Python’s built-in **immutable types are hashable,** and none of Python’s built-in mutable types are hashable.

Some of the useful methods and operations associated with dictionaries

len(d) returns the number of items in d.

d.keys() returns a view of the keys in d.

d.values() returns a view of the values in d.

k in d returns True if key k is in d.

d[k] returns the item in d with key k.

d.get(k, v) returns d[k] if k is in d, and v otherwise.

d[k] = v associates the value v with the key k in d. If there is already a value associated with k, that value is replaced.

del d[k] removes the key k from d.

for k in d iterates over the keys in d.

# Testing and Debugging

Testing is the process of running a program to try and ascertain whether or not it works as intended.

Debugging is the process of trying to fix a program that you already know does not work as intended.

Testing and debugging are not processes that you should begin to think about after a program has been built. Good programmers design their programs in ways that make them easier to test and debug. The key to doing this is breaking the program up into separate components that can be implemented, tested, and debugged independently of other components.

The first step in getting a program to work is getting the language system to agree to run it—that is, eliminating syntax errors and static semantic errors that can be detected without running the program.

Testing

The most important thing to say about testing is that its purpose is to show that bugs exist, not to show that a program is bug-free.

To quote Edsger Dijkstra, “Program testing can be used to show the presence of bugs, but never to show their absence!” Or, as Albert Einstein reputedly said, “No amount of experimentation can ever prove me right; a single experiment can prove me wrong.”

The key to testing is finding a collection of inputs, called a test suite, that has a high likelihood of revealing bugs, yet does not take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains at least one input from each partition. (Usually, constructing such a test suite is not actually possible. Think of this as an unachievable ideal.)

A partition of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets. Consider, for example, isBigger(x, y). The set of possible inputs is all pairwise combinations of integers. One way to partition this set is into these seven subsets:

x positive, y positive x negative, y negative x positive, y negative x negative, y positive x = 0, y = 0 x = 0, y ≠ 0 x ≠ 0, y = 0

If one tested the implementation on at least one value from each of these subsets, there would be reasonable probability (but no guarantee) of exposing a bug if one exists.

For most programs, finding a good partitioning of the inputs is far easier said than done. Typically, people rely on heuristics based on exploring different paths through some combination of the code and the specifications.

Heuristics based on exploring paths through the code fall into a class called glass-box testing. Heuristics based on exploring paths through the specification fall into a class called black-box testing.

Black Box Testing

In principle, black-box tests are constructed without looking at the code to be tested.

The group implementing the programs and the group testing the group are independent this independence reduces the likelihood of generating test suites that exhibit mistakes that are correlated with mistakes in the code.

Another positive feature of black-box testing is that it is robust with respect to implementation changes. Since the test data is generated without knowledge of the implementation, the tests need not be changed when the implementation is changed.

Def sqrt(x, epsilon)

There seem to be only two distinct paths through this specification: one corresponding to x = 0 and one corresponding to x > 0. And Boundary conditions should also be tested. When looking at lists, this often means looking at the empty list, a list with exactly one element, and a list containing lists. When dealing with numbers, it typically means looking at very small and very large values and combination of both as well as “typical” values.

Glass box Testing

Glass-box test suites are usually much easier to construct than black-box test suites.

The notion of a path through code is well defined, and it is relatively easy to evaluate how thoroughly one is exploring the space.

A glass-box test suite is path-complete if it exercises every potential path through the program. This is typically impossible to achieve, because it depends upon the number of times each loop is executed and the depth of each recursion.

For example, a recursive implementation of factorial follows a different path for each possible input (because the number of levels of recursion will differ). Furthermore, even a path-complete test suite does not guarantee that all bugs will be exposed.

* Exercise both branches of all if statements.
* Make sure that each except clause (see Chapter 7) is executed.
* For each for loop, have test cases in which
* The loop is not entered (e.g., if the loop is iterating over the elements of a list, make sure that it is tested on the empty list),
* The body of the loop is executed exactly once, and
* The body of the loop is executed more than once.
* For each while loop,
* Look at the same kinds of cases as when dealing with for loops.
* Include test cases corresponding to all possible ways of exiting the loop.

For example, for a loop starting with

while len(L) > 0 and not L[i] == e find cases where the loop exits because len(L) is greater than zero and cases where it exits because L[i] == e.

* For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

Conducting Tests

Testing is often thought of as occurring in two phases.

Always start with unit testing. During this phase testers construct and run tests designed to ascertain whether individual units of code (e.g., functions) work properly.

This is followed by integration testing, which is designed to ascertain whether the program as a whole behaves as intended.

In practice, testers cycle through these two phases, since failures during integration testing lead to making changes to individual units.

Integration testing is almost always more challenging than unit testing. One reason for this is that the intended behavior of an entire program is often considerably harder to characterize than the intended behavior of each of its parts.

In industry, the testing process is often highly automated. Testers40 do not sit at terminals typing inputs and checking outputs. Instead, they use test drivers that autonomously

One attraction of automating the testing process is that it facilitates regression testing. As programmers attempt to debug a program, it is all too common to install a “fix” that breaks something that used to work. Whenever any change is made, no matter how small, you should check that the program still passes all of the tests that it used to pass.

Debugging

Runtime bugs can be categorized along two dimensions:

Overt → covert:

An overt bug has an obvious manifestation, e.g., the program crashes or takes far longer (maybe forever) to run than it should.

A covert bug has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer.

Persistent → intermittent:

A persistent bug occurs every time the program is run with the same inputs.

An intermittent bug occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions. Programs that model situations in which randomness plays a role.

Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called defensive programming.

While debugging a program look for usual suspects

* Passed arguments to a function in the wrong order
* Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one
* Failed to reinitialize a variable
* Tested that two floating point values are equal (==) instead of nearly equal
* Tested for value equality (e.g., compared two lists by writing the expression L1 == L2) when you meant object equality (e.g., id(L1) == id(L2)
* Forgotten that some built-in function has a side effect
* Forgotten the () that turns a reference to an object of type function into a function invocation
* Created an unintentional alias

Stop asking yourself why the program isn’t doing what you want it to . Instead, ask yourself why it is doing what it is. That should be an easier question to answer, and will probably be a good first step in figuring out how to fix the program.

Try to explain the problem to somebody else. We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed.

Walk away, and try again tomorrow.

Safe the older version of your programs whenever you save a bug

# Exceptions And Assertions

Exceptions are everywhere. Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Test = [1, 2, 3]

Test[3]

IndexError: list index out of range

**IndexError** is the type of exception that Python raises when a program tries to access an element that is outside the bounds of an indexable type. The string following IndexError provides additional information about what caused the exception to occur.

The most commonly occurring types of exceptions are TypeError, IndexError, NameError, and ValueError.

Handling Exceptions

Up to now, we have treated exceptions as fatal events. When an exception is raised, the program terminates. When an exception is raised that causes the program to terminate, we say that an unhandled exception has been raised. An exception does not need to lead to program termination. **Exceptions, when raised, can and should be handled by the program**

If you know that a line of code might raise an exception when executed, you should handle the exception. In a well-written program, unhandled exceptions should be the exception.

successFailureRatio = numSuccesses/numFailures

print('The success/failure ratio is', successFailureRatio)

print('Now here')

This will fail if numFailures happens to be 0, The attempt to divide 0 will cause python runtime system to raise a ZeroDivisionError

You can handle the exception using try and except:

try:

successFailureRatio = numSuccesses/numFailures

print('The success/failure ratio is', successFailureRatio)

except ZeroDivisionError:

print('No failures, so the success/failure ratio is undefined.')

print('Now here')

Upon entering the try block, the interpreter attempts to evaluate the expression numSuccesses/numFailures. If expression evaluation is successful, it executes all the remaining statements in the try block and proceeds the print statement following try-except

If, however, a ZeroDivisionError exception is raised during the expression evaluation, control immediately jumps to the except block (skipping the assignment and the print statement in the try block), the print statement in the except block is executed, and then execution continues at the print statement following the try-except block.

while True:

val = input('Enter an integer: ')

try:

val = int(val)

print('The square of the number you entered is', val\*\*2)

break #to exit the while loop

except ValueError:

print(val, 'is not an integer')

No matter what the users type in it will not cause an unhandled exception

The downside of this change is that the program text has grown from two lines to eight. If there are many places where the user is asked to enter an integer, this can be problematical. This can be resolved using functions

def readInt():

while True:

val = input('Enter an integer: ')

try:

return(int(val)) #convert str to int before returning

except ValueError:

print(val, 'is not an integer')

This function can be generalized to ask for any type of input:

def readval(valtype, requestmsg, errormsg):

val = input(requestmsg + ‘ ‘)

try:

return (valtype(val))

except ValueError:

print(val, errormsg)

readVal(int, 'Enter an integer:', 'is not an integer')

The function readVal is polymorphic, i.e., it works for arguments of many different types.

Such functions are easy to write in Python, since types are firstclass objects.

If it is possible for a block of program code to raise more than one kind of exception, the reserved word except can be followed by a tuple of exceptions

except (ValueError, TypeError):

The except block will be entered if any of the listed exceptions is raised within the try block.

**Only writing except:**

Then the except block will be entered if any kind of exception is raised within the try block

Exception as a Control Flow Mechanism

Don’t think of exceptions as purely for errors. They are a convenient flow-ofcontrol mechanism that can be used to simplify programs.

In Python, it is more usual to have a function raise an exception when it cannot produce a result that is consistent with the function’s specification.

The Python raise statement forces a specified exception to occur. The form of a raise statement is

raise exceptionName(arguments)

The exceptionName is usually one of the built-in exceptions, e.g., ValueError. However, programmers can define new exceptions by creating a subclass of the built-in class Exception. Different types of exceptions can have different types of arguments, but most of the time the argument is a single string, which is used to describe the reason the exception is being raised.

Consider the function definition

def getRatios(vect1, vect2):

"""Assumes: vect1 and vect2 are equal length lists of numbers

Returns: a list containing the meaningful values of vect1[i]/vect2[i]"""

ratios = []

for index in range(len(vect1)):

try:

ratios.append(vect1[index]/vect2[index])

except ZeroDivisionError:

ratios.append(float('nan'))

except:

raise ValueError('getRatios called with bad arguments')

return ratios

There are two except blocks associated with the try block. If an exception is raised within the try block, Python first checks to see if it is a ZeroDivisionError.

If the exception is anything other than a ZeroDivisionError, the code executes the second except block, which raises a ValueError exception with an associated string.

In Theory the code should never enter the second except block due to the assumptions of the function specifications so checking this is defensive programming

try:

print(getRatios([1.0,2.0,7.0,6.0], [1.0,2.0,0.0,3.0]))

print(getRatios([], []))

print(getRatios([1.0, 2.0], [3.0]))

except ValueError as msg:

print(msg)

def getRatios(vect1, vect2): #Longer Version

"""Assumes: vect1 and vect2 are lists of equal length of numbers

Returns: a list containing the meaningful values of vect1[i]/vect2[i]"""

ratios = []

if len(vect1) != len(vect2):

raise ValueError('getRatios called with bad arguments')

for index in range(len(vect1)):

vect1Elem = vect1[index]

vect2Elem = vect2[index]

if (type(vect1Elem) not in (int, float)) or (type(vect2Elem) not in (int, float)):

raise ValueError('getRatios called with bad arguments')

if vect2Elem == 0.0:

ratios.append(float('NaN'))

else:

ratios.append(vect1Elem/vect2Elem)

return ratios

def getGrades(fname):

try:

gradesFile = open(fname, 'r')

except IOError:

raise ValueError('getGrades could not open ' + fname)

grades = []

for line in gradesFile:

try:

grades.append(float(line))

except:

raise ValueError('Unable to convert line to float')

return grades

try:

grades = getGrades('quiz1grades.txt')

grades.sort()

median = grades[len(grades)//2]

print('Median grade is', median)

except ValueError as errorMsg:

print('Whoops.', errorMsg)

Assertions

The Python assert statement provides programmers with a simple way to confirm that the state of a computation is as expected.

An assert statement can take one of two forms:

assert Boolean expression

or

assert Boolean expression, argument

When an assert statement is encountered, the Boolean expression is evaluated. If it evaluates to True, execution proceeds on its merry way. If it evaluates to False, an AssertionError exception is raised.

Assertions are a useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They are also a useful debugging tool. The can be used, for example, to confirm that intermediate values have the expected values or that a function returns an acceptable value.

# Classes And Object Oriented Programming

The key to object-oriented programming is thinking about **objects** as **collections of both data and the methods** that operate on that data.

“Objects are the core things that Python programs manipulate. Every object has a type that defines the kinds of things that programs can do with that object.”

We have relied upon **built-in** types such as **list** and **dict** and the **methods associated** with those types. But just as the designers of a programming language can build in only a small fraction of the useful functions, they can build in only a small fraction of the useful types.

**Classes are used to define or create new types**

Abstract Data Types and Classes

An abstract data type is a set of objects and the operations on those objects. These are bound together so that one can pass an object from one part of a program to another, and in doing so provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

The specifications of those operations define an interface between the abstract data type and the rest of the program. The interface defines the behavior of the operations—what they do, but not how they do it. The interface thus provides an abstraction barrier that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

**Programming** is about **managing complexity** in a way that facilitates change.

There are two powerful mechanisms available for accomplishing this: **decomposition and abstraction.**

Decomposition creates structure in a program, and abstraction suppresses detail.

The key is to suppress the appropriate details. This is where data abstraction hits the mark. One can create domain-specific types that provide a convenient abstraction. Ideally, these types capture concepts that will be relevant over the lifetime of a program.

In Python, one implements data abstractions using classes. A class definition provides a straightforward implementation of a set-of-integers abstraction called IntSet.

Notice that the docstring (the comment enclosed in """) at the top of the class definition describes the abstraction provided by the class, not information about how the class is implemented. In contrast, the comments below the docstring contain information about the implementation. This information is useful if you want to modify the implementation or build subclasses

class IntSet(object):

"""An intSet is a set of integers""" 🡪 Abstractions

#Information about the implementation (not the abstraction)

#Value of the set is represented by a list of ints, self.vals.

#Each int in the set occurs in self.vals exactly once.

def \_\_init\_\_(self):

"""Create an empty set of integers"""

self.vals = []

def insert(self, e):

"""Assumes e is an integer and inserts e into self"""

if e not in self.vals:

self.vals.append(e)

def member(self, e):

"""Assumes e is an integer

Returns True if e is in self, and False otherwise"""

return e in self.vals

def remove(self, e):

"""Assumes e is an integer and removes e from self

Raises ValueError if e is not in self"""

try:

self.vals.remove(e)

except:

raise ValueError(str(e) + ' not found')

def getMembers(self):

"""Returns a list containing the elements of self.

Nothing can be assumed about the order of the elements"""

return self.vals[:]

def \_\_str\_\_(self):

"""Returns a string representation of self"""

self.vals.sort()

result = ''

for e in self.vals:

result = result + str(e) + ','

return '{' + result[:-1] + '}' #-1 omits trailing comma

When a **function definition occurs within a class definition**, the **defined function** **is called a method** and is associated with the class. These methods are sometimes referred to as method attributes of the class.

Classes support **two kinds** of operations

* **Instantiation** is used to create instances of the class. For example, the statement s = IntSet() creates a new object of type IntSet. This object is called an instance of IntSet.
* **Attribute references** use **dot notation** to access attributes associated with the class. For example, s.member refers to the method member associated with the instance s of type IntSet.

**Each class definition begins** with the **reserved word class** followed by the **name** of the class and some information about how it relates to other classes.

Python has a number of special method names that start and end with two underscores.

**\_\_init\_\_** 🡪 Whenever a class is instantiated, a call is made to the **\_\_init\_\_** method defined in that class.

When this line of code in executed

**S = InSet()**

The interpreter will create a **new instance of type IntSet,** and then **call IntSet.\_\_init\_\_** with the newly created object as the actual parameter that is bound to the formal parameter self. When invoked, IntSet.\_\_init\_\_ creates vals, an object of type list, which becomes part of the newly created instance of type IntSet.

This list is called a data attribute of the instance of IntSet. Notice that each object of type IntSet will have a different vals list

A class should not be confused with instances of that class, just as an object of type list should not be confused with the list type. Attributes can be associated either with a class itself or with instances of a class:

* Method attributes are defined in a class definition, for example IntSet.member is an attribute of the class IntSet. When the class is instantiated, e.g., by the statement s = IntSet(), instance attributes, e.g., s.member, are created. Keep in mind that IntSet.member and s.member are different objects.
* When data attributes are associated with a class we call them class variables. When they are associated with an instance we call them instance variables. For example, vals is an instance variable because for each instance of class IntSet, vals is bound to a different list.

The representation invariant defines which values of the data attributes correspond to valid representations of class instances. The representation invariant for IntSet is that vals contains no duplicates. The implementation of \_\_init\_\_ is responsible for establishing the invariant (which holds on the empty list), and the other methods are responsible for maintaining that invariant. That is why insert appends e only if it is not already in self.vals.

\_\_str\_\_: 🡪 When the print command is used, the \_\_str\_\_ function associated with the object to be printed is automatically invoked.

If no \_\_str\_\_ method were defined executing

S = InSet()

print(S) 🡪 This will print something like <\_\_main\_\_.IntSet object at 0x1663510>

We could also print the value of s by writing print s.\_\_str\_\_() or even print IntSet.\_\_str\_\_(s)

If no \_\_eq\_\_ method is provided, all objects are considered unequal (except to themselves)

Designing Programs Using Abstract Data Types

Abstract data types are a big deal. They lead to a different way of thinking about organizing large programs. When we think about the world, we rely on abstractions. In the world of finance people talk about stocks and bonds. In the world of biology people talk about proteins and residues.

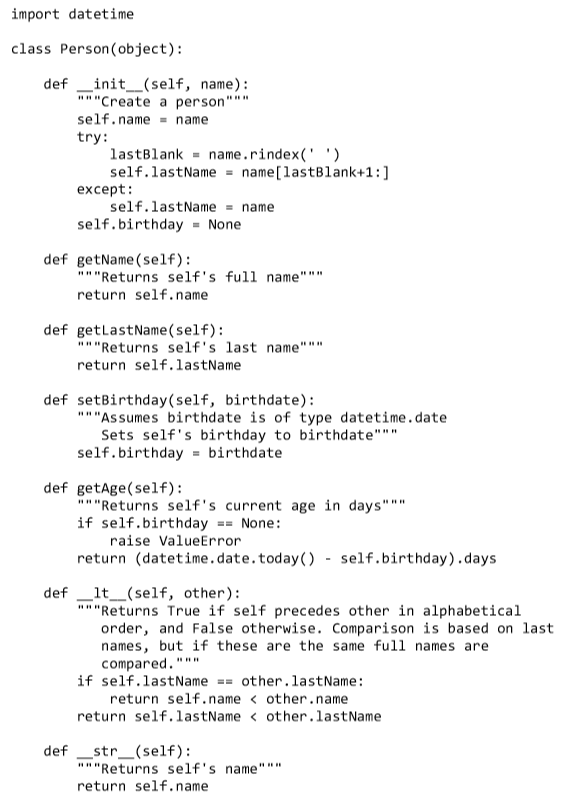
We think of bonds as having an interest rate and a maturity date as data attributes. We also think of bonds as having operations such as “set price” and “calculate yield to maturity.” Abstract data types allow us to incorporate this kind of organization into the design of programs.

Data abstraction encourages program designers to focus on the centrality of data objects rather than functions. Thinking about a program more as a collection of types than as a collection of functions leads to a profoundly different organizing principle. Among other things, it encourages one to think about programming as a process of combining relatively large chunks, since data abstractions typically encompass more functionality than do individual functions. This, in turn, leads us to think of the essence of programming as a process not of writing individual lines of code, but of composing abstractions.

Using Classes to Keep Track of Students and Faculty

As an example use of classes, imagine that you are designing a program to help keep track of all the students and faculty at a university. It is certainly possible to write such a program without using data abstraction. Each student would have a family name, a given name, a home address, a year, some grades, etc. This could all be kept in some combination of lists and dictionaries. Same can be done for faculties and staffs

Before rushing in to design a bunch of data structures, let’s think about some abstractions that might prove useful. Is there an abstraction that covers the common attributes of students, professors, and staff? Some would argue that they are all human. A class that incorporates some of the common attributes (name and birthday) of humans.



me = Person('Michael Guttag')

him = Person('Barack Hussein Obama')

her = Person('Madonna')

print(him.getLastName())

him.setBirthday(datetime.date(1961, 8, 4))

her.setBirthday(datetime.date(1958, 8, 16))

print(him.getName(), 'is', him.getAge(), 'days old')

Notice that whenever Person is instantiated an argument is supplied to the \_\_init\_\_ function. In general, when instantiating a class we need to look at the specification of the \_\_init\_\_ function for that class to know what arguments to supply and what properties those arguments should have.

After the above code is executed, there will be three instances of class Person. One can then access information about these instances using the methods associated with them. For example, him.getLastName() will return 'Obama'.

The expression him.lastName will also return 'Obama’; however, for reasons discussed later in this chapter, writing expressions that directly access instance variables is considered poor form, and should be avoided.

There is no appropriate way for a user of the Person abstraction to extract a person’s birthday, despite the fact that the implementation contains an attribute with that value.

Class Person defines yet another specially named method, \_\_lt\_\_. This method overloads the < operator. The method Person\_\_lt\_\_ gets called whenever the first argument to the < operator is of type Person. The \_\_lt\_\_ method in class Person is implemented using the binary < operator of type str. The expression self.name < other.name is shorthand for self.name.\_\_lt\_\_(other.name). Since self.name is of type str, this \_\_lt\_\_ method is the one associated with type str.

This overloading provides automatic access to any polymorphic method defined using \_\_lt\_\_. The built-in method sort is one such method. So, for example, if pList is a list composed of elements of type Person, the call pList.sort() will sort that list using the \_\_lt\_\_ method defined in class Person.

pList = [me, him, her]

for p in pList:

print(p)

pList.sort()

for p in pList:

print(p)

will first print

Michael Guttag

Barack Hussein Obama

Madonna

and then print

Michael Guttag

Madonna

Barack Hussein Obama

# Inheritance

Many types have properties in common with other types. For example, types list and str each have len functions that mean the same thing.

Inheritance provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy.

The class object is at the top of the hierarchy. This makes sense, since in Python everything that exists at run time is an object. Because Person inherits all of the properties of objects, programs can bind a variable to a Person, append a Person to a list

The class MITPerson inherits attributes from its parent class, Person, including all of the attributes that Person inherited from its parent class, object.

class MITPerson(Person):

nextIdNum = 0 #identification number

def \_\_init\_\_(self, name):

Person.\_\_init\_\_(self, name)

self.idNum = MITPerson.nextIdNum

MITPerson.nextIdNum += 1

def getIdNum(self):

return self.idNum

def \_\_lt\_\_(self, other):

return self.idNum < other.idNum

MITPerson is a subclass of Person, and therefore inherits the attributes of its superclass.

In addition to what it inherits, the subclass can:

* Add new attributes. For example, the subclass MITPerson has added the class variable nextIdNum, the instance variable idNum, and the method getIdNum.
* Override, i.e., replace, attributes of the superclass. For example, MITPerson has overridden \_\_init\_\_ and \_\_lt\_\_.

When a method has been overridden, the version of the method that is executed is based on the object that is used to invoke the method. If the type of the object is the subclass, the version defined in the subclass will be used. If the type of the object is the superclass, the version in the superclass will be used.

The method MITPerson.\_\_init\_\_ first invokes Person.\_\_init\_\_ to initialize the inherited instance variable self.name. It then initializes self.idNum, an instance variable that instances of MITPerson have but instances of Person do not.

The instance variable self.idNum is initialized using a class variable, nextIdNum, that belongs to the class MITPerson, rather than to instances of the class. When an instance of MITPerson is created, a new instance of nextIdNum is not created. This allows \_\_init\_\_ to ensure that each instance of MITPerson has a unique idNum.

p1 = MITPerson('Barbara Beaver')

print(str(p1) + '\'s id number is ' + str(p1.getIdNum()))

The first line creates a new MITPerson. The second line is a bit more complicated. When it attempts to evaluate the expression str(p1), the runtime system first checks to see if there is an \_\_str\_\_ method associated with class MITPerson. Since there is not, it next checks to see if there is an \_\_str\_\_ method associated with the superclass, Person, of MITPerson. There is, so it uses that. When the runtime system attempts to evaluate the expression p1.getidNum(), it first checks to see if there is a getIdNum method associated with class MITPerson. There is, so it invokes that method and prints

Barbara Beaver's id number is 0

p1 = MITPerson('Mark Guttag')

p2 = MITPerson('Billy Bob Beaver')

p3 = MITPerson('Billy Bob Beaver')

p4 = Person('Billy Bob Beaver')

print('p1 < p2 =', p1 < p2)

print('p3 < p2 =', p3 < p2)

print('p4 < p1 =', p4 < p1)

p1 < p2 = True

p3 < p2 = False

p4 < p1 = True

Since p1, p2, and p3 are all of type MITPerson, the interpreter will use the \_\_lt\_\_ method defined in class MITPerson when evaluating the first two comparisons

In the third comparison, the < operator is applied to operands of different types. Since the first argument of the expression is used to determine which \_\_lt\_\_ method to invoke,

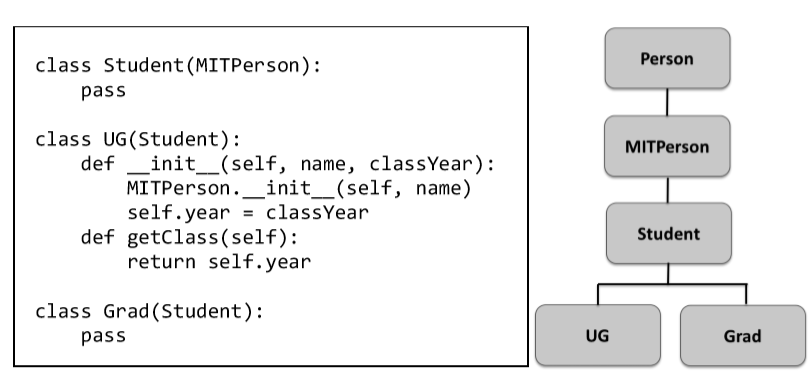
p4 < p1 is shorthand for p4.\_\_lt\_\_(p1).

Therefore, the interpreter uses the \_\_lt\_\_ method associated with the type of p4, Person, and the “people” will be ordered by name.

Print('p1 < p4 =', p1 < p4)

The runtime system will invoke the \_\_lt\_\_ operator associated with the type of p1, i.e., the one defined in class MITPerson. This will lead to the exception AttributeError: 'Person' object has no attribute 'idNum' because the object to which p4 is bound does not have an attribute idNum.

Multiple Levels of Inheritance



Adding UG seems logical, because we want to associate a year of graduation (or perhaps anticipated graduation) with each undergraduate. But what is going on with the classes Student and Grad? By using the Python reserved word pass as the body, we indicate that the class has no attributes other than those inherited from its superclass.

Why would one ever want to create a class with no new attributes? By introducing the class Grad, we gain the ability to create two different kinds of students and use their types to distinguish one kind of object from another.

p5 = Grad('Buzz Aldrin')

p6 = UG('Billy Beaver', 1984)

print(p5, 'is a graduate student is', type(p5) == Grad)

print(p5, 'is an undergraduate student is', type(p5) == UG)

will print

Buzz Aldrin is a graduate student is True

Buzz Aldrin is an undergraduate student is False

Consider going back to class MITPerson and adding the method

Consider going back to class MITPerson and adding the method

def isStudent(self):

return isinstance(self, Student)

The function isinstance is built into Python. The first argument of isinstance can be any object, but the second argument must be an object of type type. The function returns True if and only if the first argument is an instance of the second argument.

Notice that the meaning of isinstance(p6, Student) is quite different from the meaning of type(p6) == Student. The object to which p6 is bound is of type UG, not student, but since UG is a subclass of Student, the object to which p6 is bound is considered to be an instance of class Student (as well as an instance of MITPerson and Person).

Since there are only two kinds of students, we could have implemented isStudent as,

def isStudent(self):

return type(self) == Grad or type(self) == UG

However, if a new type of student were introduced at some later point it would be necessary to go back and edit the code implementing isStudent. By introducing the intermediate class Student and using isinstance we avoid this problem.

class TransferStudent(Student):

def \_\_init\_\_(self, name, fromSchool):

MITPerson.\_\_init\_\_(self, name)

self.fromSchool = fromSchool

def getOldSchool(self):

return self.fromSchool

The Substitution Principle

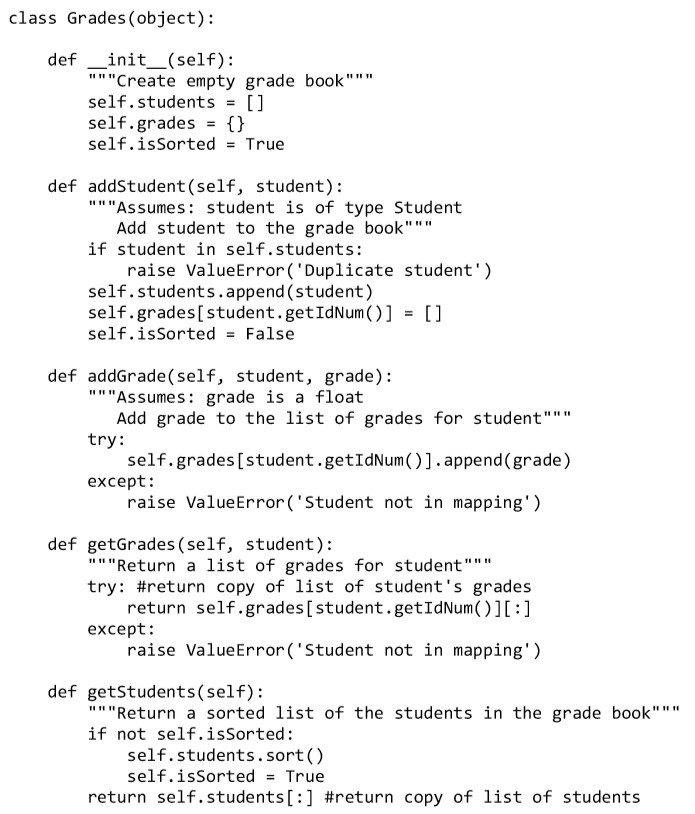
When subclassing is used to define a type hierarchy, the subclasses should be thought of as extending the behavior of their superclasses. We do this by adding new attributes or overriding attributes inherited from a superclass.

For example, TransferStudent extends Student by introducing a former school.

Sometimes, the subclass overrides methods from the superclass, but this must be done with care. In particular, important behaviors of the supertype must be supported by each of its subtypes. If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype.

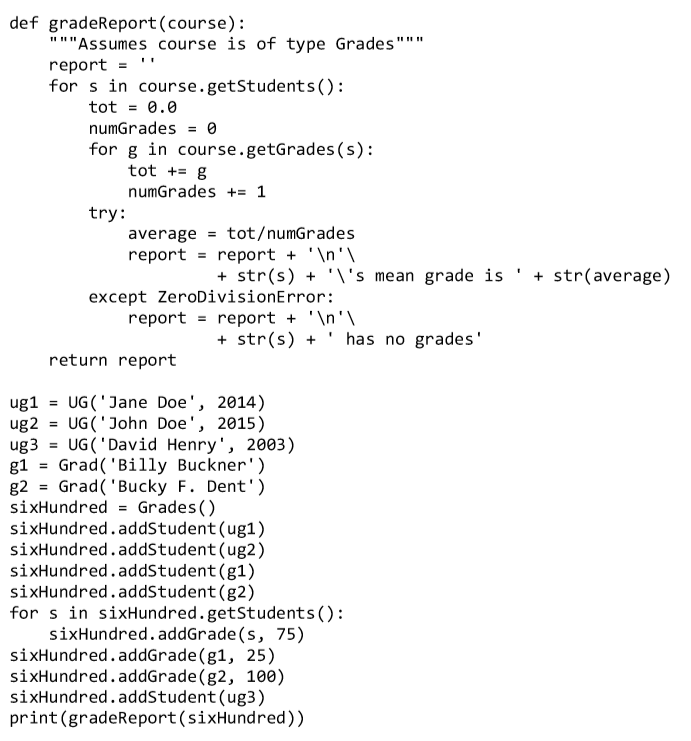
For example, it should be possible to write client code using the specification of Student and have it work correctly on a TransferStudent. But the other way won’t work

Encapsulation and Information Hiding



The instance variable isSorted is used to keep track of whether or not the list of students has been sorted since the last time a student was added to it. This allows the implementation of getStudents to avoid sorting an already sorted list.

A function that uses class Grades to produce a grade report for some students taking the course named C 1



When run, the code in the figure prints

Jane Doe's mean grade is 75.0

John Doe's mean grade is 75.0

David Henry has no grades

Billy Buckner's mean grade is 50.0

Bucky F. Dent's mean grade is 87.5

There are two important concepts at the heart of object-oriented programming.

* The first is the idea of encapsulation. By this we mean the bundling together of data attributes and the methods for operating on them.

For example, if we write Rafael = MITPerson('Rafael Reif')

we can use dot notation to access attributes such as Rafael’s name and identification number.

* The second important concept is information hiding. This is one of the keys to modularity. If those parts of the program that use a class (i.e., the clients of the class) rely only on the specifications of the methods in the class, a programmer implementing the class is free to change the implementation of the class (e.g., to improve efficiency) without worrying that the change will break code that uses the class.

Some programming languages (Java and C++, for example) provide mechanisms for enforcing information hiding. Programmers can make the attributes of a class private, so that clients of the class can access the data only through the object's methods.

Python 3 uses a naming convention to make attributes invisible outside the class. When the name of an attribute starts with \_\_ but does not end with \_\_, that attribute is not visible outside the class.

class infoHiding(object):

def \_\_init\_\_(self):

self.visible = 'Look at me'

self.\_\_alsoVisible\_\_ = 'Look at me too'

self.\_\_invisible = 'Don\'t look at me directly'

def printVisible(self):

print(self.visible)

def printInvisible(self):

print(self.\_\_invisible)

def \_\_printInvisible(self):

print(self.\_\_invisible)

def \_\_printInvisible\_\_(self):

print(self.\_\_invisible)

test = infoHiding()

print(test.visible) 🡪 Look at me

print(test.\_\_alsoVisible\_\_) 🡪 Look at me

print(test.\_\_invisible) 🡪 Error ‘infoHiding object has no attribute ‘\_\_invisible’

test = infoHiding()

test.printInvisible() 🡪 Don’t look at me directly

test.\_\_printInvisible\_\_() 🡪 Don’t look at me directly

test.\_\_printInvisible() 🡪 Error: ‘infoHiding’ object has no attribute

‘\_\_printInvisible’

class subClass(infoHiding):

def \_\_init\_\_(self):

print('from subclass', self.\_\_invisible)

testSub = subClass()

prints

Error: 'subClass' object has no attribute '\_subClass\_\_invisible'

Notice that when a subclass attempts to use a hidden attribute of its superclass an AttributeError occurs. This can make using information hiding in Python a bit cumbersome.

Client of class Person can write the expression Rafael.lastName rather than Rafael.getLastName(). This is unfortunate because it allows the client code to rely upon something that is not part of the specification of Person, and is therefore subject to change. If the implementation of Person were changed, for example to extract the last name whenever it is requested rather than store it in an instance variable, then the client code would no longer work.

Python let programs read instance and class variables from outside the class definition, it also lets programs write these variables. So, for example, the code

Rafael.birthday = '8/21/50' is perfectly legal. This would lead to a runtime type error, were Rafael.getAge invoked later in the computation. It is even possible to create instance variables from outside the class definition. For example, Python will not complain if the assignment statement me.age = Rafael.getIdNum() occurs outside the class definition.

While this relatively weak static semantic checking is a flaw in Python, it is not a fatal flaw. A disciplined programmer can simply follow the sensible rule of not directly accessing data attributes from outside the class in which they are defined

Generators

A perceived risk of information hiding is that preventing client programs from directly accessing critical data structures leads to an unacceptable loss of efficiency.

In the early days of data abstraction, many were concerned about the cost of introducing extraneous function/method calls. Modern compilation technology makes this concern moot.

A more serious issue is that client programs will be forced to use inefficient algorithms.

Consider the implementation of gradeReport, The invocation of course.getStudents creates and returns a list of size n, where n is the number of students. This is probably not a problem for a grade book for a single class, but imagine keeping track of the grades of 1.7 million high school students taking the SATs. Creating a new list of that size when the list already exists is a significant inefficiency. One solution is to abandon the abstraction and allow gradeReport to directly access the instance variable course.students, but that would violate information hiding. Fortunately, there is a better solution.

def getStudents(self):

"""Return the students in the grade book one at a time

in alphabetical order"""

if not self.isSorted:

self.students.sort()

self.isSorted = True

for s in self.students:

yield s

Any function definition containing a yield statement is treated in a special way. The presence of yield tells the Python system that the function is a generator. Generators are typically used in conjunction with for statements as in

for s in course.getStudents():

At the start of the first iteration of a for loop that uses a generator, the generator is invoked and runs until the first time a yield statement is executed, at which point it returns the value of the expression in the yield statement. On the next iteration, the generator resumes execution immediately following the yield, with all local variables bound to the objects to which they were bound when the yield statement was executed, and again runs until a yield statement is executed. It continues to do this until it runs out of code to execute or executes a return statement, at which point the loop is exited

Generating one value at a time will be more efficient, because a new list containing the students will not be created.

# A Simplistic Introduction to Algorithmic Complexity

The most important thing to think about when designing and implementing a program is that it should produce results that can be relied upon.

A program that warns airplanes of potential obstructions needs to issue the warning before the obstructions are encountered. Performance can also affect the utility of many non-realtime programs. The number of transactions completed per minute is an important metric when evaluating the utility of database systems. Users care about the time required to start an application on their phone. Biologists care about how long their phylogenetic inference calculations take.

Writing efficient programs is not easy. The most straightforward solution is often not the most efficient. Computationally efficient algorithms often employ subtle tricks that can make them difficult to understand. Consequently, programmers often increase the conceptual complexity of a program in an effort to reduce its computational complexity.

Thinking about Computational Complexity

How long will the following function take to run?

def f(i):

"""Assumes i is an int and i >= 0"""

answer = 1

while i >= 1:

answer \*= i

i -= 1

return answer

We could run the program on some input and time it. But that wouldn’t be particularly informative because the result would depend upon

• the speed of the computer on which it is run,

• the efficiency of the Python implementation on that machine, and

• the value of the input

We get around the first two issues by using a more abstract measure of time. Instead of measuring time in milliseconds, we measure time in terms of the number of basic steps executed by the program. We assume **step** is an operation that takes a fixed amount of time, such as binding a variable to an object, making a comparison, executing an arithmetic operation, or accessing an object in memory.

Now that we have a more abstract way to think about the meaning of time, we turn to the question of dependence on the value of the input. We deal with that by moving away from expressing time complexity as a single number and instead relating it to the sizes of the inputs. This allows us to compare the efficiency of two algorithms by talking about how the running time of each grows with respect to the sizes of the inputs.

The actual running time of an algorithm depends not only upon the sizes of the inputs but also upon their values. Consider, for example, the linear search algorithm implemented by def linearSearch(L, x):

for e in L:

if e == x:

return True

return False

Suppose that L is a list containing a million elements, and consider the call

linearSearch(L, 3). If the first element in L is 3, linearSearch will return True almost immediately. On the other hand, if 3 is not in L, linearSearch will have to examine all one million elements before returning False.

In general, there are three broad cases to think about:

* The best-case running time is the running time of the algorithm when the inputs are as favorable as possible. I.e., the best-case running time is the minimum running time over all the possible inputs of a given size.
* The worst-case running time is the maximum running time over all the possible inputs of a given size.
* The average-case (also called expected-case) running time is the average running time over all possible inputs of a given size.

People usually focus on the worst case. All engineers share a common article of faith, Murphy’s Law: If something can go wrong, it will go wrong. The worstcase provides an upper bound on the running time. This is critical in situations where there is a time constraint on how long a computation can take.

Let’s look at the worst-case running time of an iterative implementation of the factorial function:

def fact(n):

"""Assumes n is a natural number Returns n!"""

answer = 1

while n > 1:

answer \*= n

n -= 1

return answer

The number of steps required to run this program is something like 2 (1 for the initial assignment statement and 1 for the return) + 5n (counting 1 step for the test in the while, 2 steps for the first assignment statement in the while loop, and 2 steps for the second assignment statement in the loop). So, for example, if n is 1000, the function will execute roughly 5002 steps.

As n get larger we typically ignore the additive constants when reasoning about running time. Multiplicative constants are more problematical

Asymptotic Notation

We use asymptotic notation to provide a formal way to talk about the relationship between the running time of an algorithm and the size of its inputs.

The underlying motivation is that almost any algorithm is sufficiently efficient when run on small inputs. What we typically need to worry about is the efficiency of the algorithm when run on very large inputs.

As a proxy for “very large,” asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity.

def f(x):

"""Assume x is an int > 0"""

ans = 0

#Loop that takes constant time

for i in range(1000):

ans += 1

print('Number of additions so far', ans)

#Loop that takes time x

for i in range(x):

ans += 1

print('Number of additions so far', ans)

#Nested loops take time x\*\*2

for i in range(x):

for j in range(x):

ans += 1

ans += 1

print('Number of additions so far', ans)

return ans

If one assumes that each line of code takes one unit of time to execute, the running time of this function can be described as 1000 + x + 2X2. The constant 1000 corresponds to the number of times the first loop is executed. The term x corresponds to the number of times the second loop is executed. Finally, the term 2x2 corresponds to the time spent executing the two statements in the nested for loop.

The Call f(10) will print

Number of additions so far 1000

Number of additions so far 1010

Number of additions so far 1210

The Call f(1000) will print

Number of additions so far 1000

Number of additions so far 2000

Number of additions so far 2002000

For small values of x the constant term dominates, but as x gets large the constant term and linear term becomes negligible

Clearly, we can get a meaningful notion of how long this code will take to run on very large inputs by considering only the inner loop, i.e., the quadratic component.

This kind of analysis leads us to use the following rules of thumb in describing the asymptotic complexity of an algorithm:

• If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others.

• If the remaining term is a product, drop any constants.

The most commonly used asymptotic notation is called “Big O” notation.54 Big O notation is used to give an upper bound on the asymptotic growth (often called the order of growth) of a function.

# Some Important Complexity Classes

Some of the most common instances of Big O are listed below. In each case, n is a measure of the size of the inputs to the function.

• O(1) denotes constant running time.

• O(log n) denotes logarithmic running time.

• O(n) denotes linear running time.

• O(n log n) denotes log-linear running time.

• O(nk) denotes polynomial running time. Notice that k is a constant.

• O(cn) denotes exponential running time. Here a constant is being raised to a power based on the size of the input.

# Some Simple Algorithms and Data Structures

The goal of this chapter is to help you develop some general intuitions about how to approach questions of efficiency. By the time you get through this chapter you should understand why some programs complete in the blink of an eye, why some need to run overnight, and why some wouldn’t complete in your lifetime.

At first we used brute force exhaustive enumeration for finding an approximation to the roots of the polynomial but this wasn’t efficient and practical since search space was large to make brute force, This led us to consider more efficient algorithms such bisection search and Newton Raphson.

The major point was the key of efficiency is a good algorithm, not clever coding tricks.

In the sciences (physical, life, and social), programmers often start by quickly coding up a simple algorithm to test the plausibility of a hypothesis about a data set, and then run it on a small amount of data. If this yields encouraging results, the hard work of producing an implementation that can be run (perhaps over and over again) on large data sets begins. Such implementations need to be based on efficient algorithms.

Efficient algorithms are hard to implement so what we do instead is learn to reduce the most complex aspects of the problems we are faced with to previously solved problems

More specifically, we

• Develop an understanding of the inherent complexity of the problem,

• Think about how to break that problem up into subproblems, and

• Relate those subproblems to other problems for which efficient algorithms already exist.

It is often a good strategy to start by solving the problem at hand in the most straightforward manner possible, instrument it to find any computational bottlenecks, and then look for ways to improve the computational complexity of those parts of the program contributing to the bottlenecks.

Search Algorithms

A search algorithm is a method for finding an item or group of items with specific properties within a collection of items. We refer to the collection of items as a search space.

The search space might be concrete, such as a set of electronics medical records, or something abstract, such as a set of all integers.

def search(L, e):

"""Assumes L is a list.

Returns True if e is in L and False otherwise"""

Linear Search and Using Indirection to Access Elements

Python uses the following algorithm to determine if an element is in a list:

for i in range(len(L)):

if L[i] == e:

return True

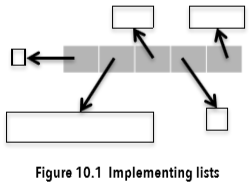
return False

If the element e is not in the list the algorithm will perform O(len(L)) tests, i.e., the complexity is at best linear in the length of L.

Let’s start by considering the simple case where each element of the list is an integer. This implies that each element of the list is the same size, e.g., four units of memory (four eight-bit bytes). Assuming that the elements of the list are stored contiguously, the address in memory of the ith element of the list is simply start + 4\*i, where start is the address of the start of the list. Therefore we can assume that Python could compute the address of the ith element of a list of integers in constant time.

Of course, we know that Python lists can contain objects of types other than int, and that the same list can contain objects of many different types and sizes. You might think that this would present a problem, but it does not.

In Python, a list is represented as a length (the number of objects in the list) and a sequence of fixed-size pointers to objects. Figure 10.1 illustrates the use of these pointers. The shaded region represents a list containing four elements. The leftmost shaded box contains a pointer to an integer indicating the length of the list. Each of the other shaded boxes contains a pointer to an object in the list.



If the length field occupies four units of memory, and each pointer (address) occupies four units of memory, the address of the ith element of the list is stored at the address

Start + 4 + 4\*i. Again, this address can be found in constant time, and then the value stored at that address can be used to access the ith element. This access too is a constant-time operation.

Generally speaking, indirection involves accessing something by first accessing something else that contains a reference to the thing initially sought. This is what happens each time we use a variable to refer to the object to which that variable is bound. When we use a variable to access a list and then a reference stored in that list to access another object, we are going through two levels of indirection.

Binary Search and Exploiting Assumptions

Implementing search(L, e) is O(len(L)) the best we can do?

Yes, if we know nothing about the relationship of the values of the elements in the lists and the order in which they are stored. In the worst case, we have to look at each element in L to determine whether L contains e.

But suppose we know something about the order in which elements are stored, e.g., suppose we know that we have a list of integers stored in ascending order. We could change the implementation so that the search stops when it reaches a number larger than the number for which it is searching

def search(L, e):

"""Assumes L is a list, the elements of which are in

ascending order.

Returns True if e is in L and False otherwise"""

for i in range(len(L)):

if L[i] == e:

return True

if L[i] > e:

return False

return False

This would improve the average running time. However, it would not change the worst-case complexity of the algorithm, We can get a considerable improvement in the worst case complexity by using an algorithm binary search, and rely on the assumption that list is ordered

The idea is simple:

1. Pick an index, i, that divides the list L roughly in half.

2. Ask if L[i] == e.

3. If not, ask whether L[i] is larger or smaller than e.

4. Depending upon the answer, search either the left or right half of L for e.

The most straightforward implementation of binary search uses recursion.

def search(L, e):

"""Assumes L is a list, the elements of which are in

ascending order.

Returns True if e is in L and False otherwise"""

def bSearch(L, e, low, high):

#Decrements high – low

if high == low:

return L[low] == e

mid = (low + high)//2

if L[mid] == e:

return True

elif L[mid] > e:

if low == mid: #nothing left to search

return False

else:

return bSearch(L, e, low, mid - 1)

else:

return bSearch(L, e, mid + 1, high)

if len(L) == 0:

return False

else:

return bSearch(L, e, 0, len(L) - 1)

The specification says that the implementation may assume that L is sorted in ascending order. The burden of making sure that this assumption is satisfied lies in with caller of searech

Should search be modified to check that the assumption is satisfied?

This might eliminate a source of errors, but it would defeat the purpose of using binary search, since checking the assumption would itself take O(len(L)) time.

Functions such as search are often called wrapper functions. The function provides a nice interface for client code, but is essentially a pass-through that does no serious computation. Instead, it calls the helper function bSearch with appropriate arguments. This raises the question of why not eliminate search and have clients call bSearch directly? The reason is that the parameters low and high have nothing to do with the abstraction of searching a list for an element. They are implementation details that should be hidden from those writing programs that call search.