COMP7023 Python Programming: Part I

Olaf Chitil

Autumn 2025
Version: September 26, 2025

Contents

[1] https://creativecommons.org/licenses/by-nc/4.0/
[2] https://composingprograms.com
[3] http://denero.org
[4] https://web.mit.edu/6.001/6.037/sicp.pdf
[5] https://mitpress.mit.edu/9780262510875/structure-and-interpretation-of-computer-programs/

Introduction

Computer science is a tremendously broad academic discipline. The areas of globally distributed systems, artificial intelligence, data science, robotics, graphics, security, scientific computing, computer architecture, and dozens of emerging sub-fields all expand with new techniques and discoveries every year. The rapid progress of computer science has left few aspects of human life unaffected. Commerce, communication, science, art, leisure, and politics have all been reinvented as computational domains.

The high productivity of computer science is only possible because the discipline is built upon an elegant and powerful set of fundamental ideas. All computing begins with representing information, specifying logic to process it, and designing abstractions that manage the complexity of that logic. Mastering these fundamentals will require us to understand precisely how computers interpret computer programs and carry out computational processes.

Programming Languages

Computers are *general-purpose* machines. All other machines, such as washing-machines, microwaves, cars and clocks, are each designed for one particular purpose. A car cannot suddenly become a washing machine. In contrast, a computer can be used for analysing data, controlling a robot, communicating with people far away, playing games and infinitely many other purposes. Computers are general-purpose because they are programmable. A computer follows the instructions of a program, which can easily be replaced by a different program for a different purpose. The term *software* expresses the simple replaceability of programs (and data).

The central processing unit (CPU) of a computer executes the instructions of a *machine language*. These instructions are relatively primitive (e.g. arithmetic on a few bytes of data stored at specific memory addresses) and hard for humans to read and write. Machine programs tend to be long and time-consuming for humans to design.

Therefore, most programs are written in *high-level programming languages*. High-level programming languages were designed to be close to human language (mostly English and established mathematical notation) and to be easy for humans to read and write. High-level programming languages include constructs to organise a program such that the human programmer can write correct programs for complex purposes more easily.

Because the computer itself only understands a machine language, any program in a high-level programming language needs to be translated into a machine language. Programs for this translation purpose have already been built: a *compiler* translates a whole high-level program into a machine language program before that is run; an *interpreter* translates a high-level program into machine language bit by bit, interleaved with running these machine language program bits; it works like a simultaneous interpreter of

human languages.

Since the 1950's hundreds of high-level programming languages have been developed; however, only about 20–50 are in wide-spread use. The majority of these hundreds of languages were experimental, yielding insight into language design. New language concepts and constructs were developed over time. Some once widely used programming languages nearly died out (e.g. Pascal), while others evolved. The oldest high-level programming language Fortran (first released in 1957) is still widely used in science and engineering, but has substantially changed over time.

Programming in Python

These notes maintain the spirit of SICP[6] by introducing programming language features in step with techniques for abstraction and a rigorous model of computation. We will work primarily with the Python language[7].

Python is a widely used programming language that has recruited enthusiasts from many professions: data analysts, web programmers, game engineers, scientists, academics, and even designers of new programming languages. When you learn Python, you join a million-person-strong community of developers. Developer communities are tremendously important institutions: members help each other solve problems, share their projects and experiences, and collectively develop software and tools.

The Python language was conceived and first implemented by Guido van Rossum[8] in the late 1980's. The language is the product of a large volunteer community[9].

Python supports a variety of different programming styles or paradigms (structured, object-oriented, functional), which we will explore.

Installing Python 3

The best way to get started programming in Python is to interact with the interpreter directly. This section describes how to install Python 3, initiate an interactive session with the interpreter, and start programming.

Python has many versions. This text will use the most recent stable version of Python 3. Many computers have older versions of Python installed already, such as Python 2.7, but those will not match the descriptions in this text. You should be able to use any computer, but expect to install Python 3. (Don't worry, Python is free.)

[6] https://mitpress.mit.edu/9780262510875/structure-and-interpretation-of-computer-programs/
[7] https://docs.python.org/
[8] https://en.wikipedia.org/wiki/Guido_van_Rossum
[9] https://www.python.org/psf/fellows-roster/

You can download Python 3 from the Python webpage[10] by clicking on the version that begins with 3 (not 2). Follow the instructions of the installer to complete installation.

For further guidance, try these video tutorials on Windows installation[11] and Mac installation[12] of Python 3, created for course CS61A at Berkley University.

Interactive Sessions

In an interactive Python session, you type some Python code after the prompt, `>>>`. The Python interpreter reads and executes what you type, carrying out your various commands.

To start an interactive session, run the Python 3 interpreter. Type `python3` at a terminal prompt (Mac/Unix/Linux) or open the Python 3 application in Windows.

If you see the Python prompt, `>>>`, then you have successfully started an interactive session. These notes depict example interactions using the prompt, followed by some input.

```
>>> 2 + 2
4
```

[10] https://www.python.org
[11] https://www.youtube.com/watch?v=54-wuFsPi0w
[12] https://www.youtube.com/watch?v=smHuBHxJdK8

Part I

Building Abstractions with Functions

A programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organise our ideas about computational processes. Programs serve to communicate those ideas among the members of a programming community. Thus, programs must be written for people to read, and only incidentally for machines to execute.

When we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three such mechanisms:

- **primitive expressions and statements**, which represent the simplest building blocks that the language provides,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units; abstraction hides details of the compound elements and thus keeps complexity manageable.

In programming, we deal with two kinds of elements: functions and data. (Soon we will discover that they are really not so distinct.) Informally, data is stuff that we want to manipulate, and functions describe the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive functions, as well as have some methods for combining and abstracting both functions and data.

1 The Python Calculator

Having experimented with the full Python interpreter in the previous section, we now start anew, methodically exploring the Python language. Be patient if the examples seem simplistic — more exciting material is soon to come.

1.1 Expressions

We begin with primitive expressions. One kind of primitive expression is a number. More precisely, the expression that you type consists of the numerals that represent the number in base 10.

```
>>> 42
42
```

Expressions representing numbers may be combined with mathematical operators to form a compound expression, which the interpreter will evaluate:

```
>>> 2 * 4
8
>>> -1 - -1
0
>>> 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
0.9921875
```

These mathematical expressions use infix notation, where the operator (e.g., `+`, `-`, `*`, or `/`) appears in between the operands (numbers). Python includes many ways to form compound expressions. Rather than attempt to enumerate them all immediately, we will introduce new expression forms as we go, along with the language features that they support.

1.2 Errors

The Python interpreter gives an error message, if you input something invalid.

```
>>> 2 * 4) + 3
  File "<python-input-4>", line 1
    2 * 4) + 3
         ^
SyntaxError: unmatched ')'
>>> 1/2 + 1/
  File "<python-input-5>", line 1
    1/2 + 1/
            ^
SyntaxError: invalid syntax
```

The error message in the last line gives a short explanation of the problem.

1.3 Call Expressions

The most important kind of compound expression is a call expression, which applies a function to some arguments. Recall from algebra that the mathematical notion of a function is a mapping from some input arguments to an output value. For instance,

the max function maps its inputs to a single output, which is the largest of the inputs. The way in which Python expresses function application is the same as in conventional mathematics.

```
>>> max(7.5, 9.5)
9.5
```

This call expression has subexpressions: the operator is an expression that precedes parentheses, which enclose a comma-delimited list of operand expressions.

The operator specifies a function. When this call expression is evaluated, we say that the function max is called with arguments 7.5 and 9.5, and returns a value of 9.5.

The order of the arguments in a call expression matters. For instance, the function pow raises its first argument to the power of its second argument.

```
>>> pow(100, 2)
10000
>>> pow(2, 100)
1267650600228229401496703205376
```

Function notation has three principal advantages over the mathematical convention of infix notation. First, functions may take an arbitrary number of arguments:

```
>>> max(1, -2, 3, -4)
3
```

No ambiguity can arise, because the function name always precedes its arguments.

Second, function notation extends in a straightforward way to nested expressions, where the elements are themselves compound expressions. In nested call expressions, unlike compound infix expressions, the structure of the nesting is entirely explicit in the parentheses.

```
>>> max(min(1, -2), min(pow(3, 5), -4))
-2
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Python interpreter can evaluate. However, humans quickly get confused by multi-level nesting. An important role for you as a programmer is to structure expressions so that they remain interpretable by yourself, your programming partners, and other people who may read your expressions in the future.

Third, mathematical notation has a great variety of forms: multiplication appears between terms, exponents appear as superscripts, division as a horizontal bar, and a square root as a roof with slanted siding. Some of this notation is very hard to type! However, all of this complexity can be unified via the notation of call expressions. While Python supports common mathematical operators using infix notation (like + and -), any operator can be expressed as a function with a name.

1.4 Importing Library Functions

Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make all of their names available by default. Instead, it organises the functions and other quantities that it knows about into modules, which together comprise the Python Library. To use these elements, one imports them. For example, the `math` module provides a variety of familiar mathematical functions:

```
>>> from math import sqrt
>>> sqrt(256)
16.0
```

and the `operator` module provides access to functions corresponding to infix operators:

```
>>> from operator import add, sub, mul
>>> add(14, 28)
42
>>> sub(100, mul(7, add(8, 4)))
16
```

An import statement designates a module name (e.g., `operator` or `math`), and then lists the named attributes of that module to import (e.g., `sqrt`). Once a function is imported, it can be called multiple times.

There is no difference between using these operator functions (e.g., `add`) and the operator symbols themselves (e.g., `+`). Conventionally, most programmers use symbols and infix notation to express simple arithmetic.

The Python 3 Library Docs[13] list the functions defined by each module, such as the math module. However, this documentation is written for developers who know the whole language well. For now, you may find that experimenting with a function tells you more about its behaviour than reading the documentation. As you become familiar with the Python language and vocabulary, this documentation will become a valuable reference source.

[13] https://docs.python.org/library/

In general, programmers do not want to reinvent the wheel. In addition to the standard Python library there exists a large number of Python libraries with numerous modules, for example `NumPy` for numerical computations and matrix operations, `pandas` for analysing spreadsheet-like data, `matplotlib` for plotting diagrams from data, `TensorFlow` for deep learning and `Django` for web development. Much effort has been put into designing these libraries, ensuring that they are documented, correct and efficient. In practice, most programs use such libraries extensively. Here we focus on the principles of programming, which will enable us to later use these libraries.

1.5 Names and the Environment

A critical aspect of a programming language is the means it provides for using *names* to refer to computational objects. If a value has been given a name, we say that the name *binds* to the value.

In Python, we can establish new bindings using the *assignment statement*, which contains a name to the left of = and a value to the right:

```
>>> radius = 10
>>> radius
10
>>> 2 * radius
20
```

Names are also bound via *import statements*.

```
>>> from math import pi
>>> pi * 71 / 223
1.0002380197528042
```

The = symbol is called the assignment operator in Python (and many other languages). Assignment is our simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations, such as the area computed above. In this way, complex programs are constructed by building, step by step, computational objects of increasing complexity.

The possibility of binding names to values and later retrieving those values by name means that the interpreter must maintain some sort of memory that keeps track of the names, values, and bindings. This memory is called an *environment*.

Names can also be bound to functions. For instance, the name max is bound to the max function we have been using. Functions, unlike numbers, are tricky to render as text, so Python prints an identifying description instead, when asked to describe a function:

```
>>> max
<built-in function max>
```

We can use assignment statements to give new names to existing functions.

```
>>> f = max
>>> f
<built-in function max>
>>> f(2, 3, 4)
4
```

And successive assignment statements can rebind a name to a new value.

```
>>> f = 2
>>> f
2
```

In Python, names are often called variable names or *variables* because they can be bound to different values in the course of executing a program. When a name is bound to a new value through assignment, it is no longer bound to any previous value. One can even bind built-in names to new values.

```
>>> max = 5
>>> max
5
```

After assigning max to 5, the name max is no longer bound to a function, and so attempting to call max(2, 3, 4) will cause an error.

When executing an assignment statement, Python evaluates the expression to the right of = before changing the binding to the name on the left. Therefore, one can refer to a name in right-side expression, even if it is the name to be bound by the assignment statement.

```
>>> x = 2
>>> x = x + 1
>>> x
3
```

We can also assign multiple values to multiple names in a single statement, where names on the left of = and expressions on the right of = are separated by commas.

```
>>> area, circumference = pi * radius * radius, 2 * pi * radius
>>> area
314.1592653589793
>>> circumference
62.83185307179586
```

Changing the value of one name does not affect other names. Below, even though the name `area` was bound to a value defined originally in terms of `radius`, the value of `area` has not changed. Updating the value of `area` requires another assignment statement.

```
>>> radius = 11
>>> area
314.1592653589793
>>> area = pi * radius * radius
380.132711084365
```

With multiple assignment, all expressions to the right of `=` are evaluated before any names to the left are bound to those values. As a result of this rule, swapping the values bound to two names can be performed in a single statement.

```
>>> x, y = 3, 4.5
>>> y, x = x, y
>>> x
4.5
>>> y
3
```

1.6 Evaluating Nested Expressions

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating nested call expressions, the interpreter is itself following a procedure.

To *evaluate* a call expression, Python will do the following:

1. Evaluate the operator and operand subexpressions, then
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

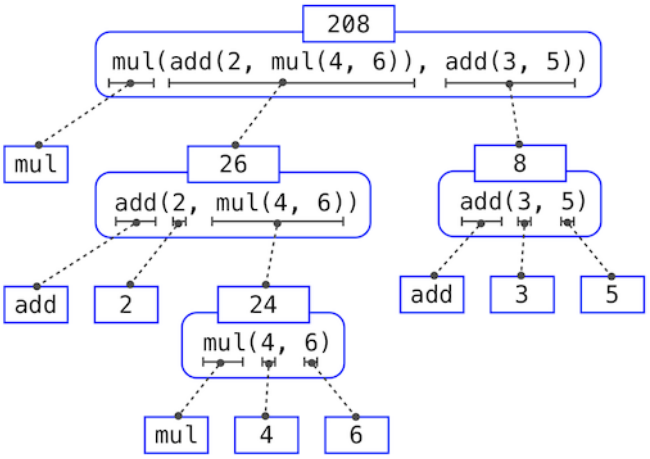
Even this simple procedure illustrates some important points about processes in general. The first step dictates that in order to accomplish the evaluation process for a call

expression we must first evaluate other expressions. Thus, the evaluation procedure is recursive in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.

For example, evaluating

```
>>> sub(pow(2, add(1, 10)), pow(2, 5))
2016
```

requires that this evaluation procedure be applied four times. If we draw each expression that we evaluate, we can visualise the hierarchical structure of this process.



This illustration is called an expression tree. In computer science, trees are conventionally displayed with the root at the top growing downwards. The objects at each point in a tree are called nodes; in this case, they are expressions paired with their values.

Evaluating its root, the full expression at the top, requires first evaluating the branches that are its subexpressions. The leaf expressions (that is, nodes with no branches stemming from them) represent either functions or numbers. The interior nodes have two parts: the call expression to which our evaluation rule is applied, and the result of that expression. Viewing evaluation in terms of this tree, we can imagine that the values of the operands percolate upward, starting from the terminal nodes and then combining at higher and higher levels.

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not call expressions, but primitive expressions such as numerals (e.g., 2) and names (e.g., add). We take care of the primitive cases by stipulating that

- A numeral evaluates to the number it names,

- A name evaluates to the value associated with that name in the current environment.

Notice the important role of an environment in determining the meaning of the symbols in expressions. In Python, it is meaningless to speak of the value of an expression such as

```
>>> add(x, 1)
```

without specifying any information about the environment that would provide a meaning for the name `x` (or even for the name `add`). Environments provide the context in which evaluation takes place, which plays an important role in our understanding of program execution.

The result of evaluating an expression is called a *value* or *object*.

This evaluation procedure does not suffice to evaluate all Python code, only call expressions, numerals, and names. For instance, it does not handle assignment statements. Executing

```
>>> x = 3
```

does not return a value nor evaluate a function on some arguments, since the purpose of assignment is instead to bind a name to a value. In general, statements are not *evaluated* but *executed*; they do not produce a value but instead make some change. Each type of expression or statement has its own evaluation or execution procedure.

A pedantic note: when we say that "a numeral evaluates to a number," we actually mean that the Python interpreter evaluates a numeral to a number. It is the interpreter which endows meaning to the programming language. Given that the interpreter is a fixed program that always behaves consistently, we can say that numerals (and expressions) themselves evaluate to values in the context of Python programs.

1.7 The Non-Pure Print Function

Throughout this text, we will distinguish between two types of functions.

Pure functions. Functions have some input (their arguments) and return some output (the result of applying them). The built-in function `abs` is pure.

```
>>> abs(-2)
2
```

Pure functions have the property that applying them has no effects beyond returning a value. Moreover, a pure function must always return the same value when called twice with the same arguments.

Non-pure functions. In addition to returning a value, applying a non-pure function can generate side effects, which make some change to the state of the interpreter or computer. A common side effect is to generate additional output beyond the return value, using the print function.

```
>>> print(1, 2, 3)
1 2 3
```

While `print` and `abs` may appear to be similar in these examples, they work in fundamentally different ways. The value that print returns is always `None`, a special Python value that represents nothing. The interactive Python interpreter does not automatically print the value `None`. In the case of print, the function itself is printing output as a side effect of being called.

A nested expression of calls to print highlights the non-pure character of the function.

```
>>> print(print(1), print(2))
1
2
None None
```

If you find this output to be unexpected, draw an expression tree to clarify why evaluating this expression produces this peculiar output.

Be careful with print! The fact that it returns `None` means that it should not be the expression in an assignment statement.

```
>>> two = print(2)
2
>>> print(two)
None
```

Pure functions are restricted in that they cannot have side effects or change behavior over time. Imposing these restrictions yields substantial benefits. First, pure functions can be composed more reliably into compound call expressions. We can see in the non-pure function example above that `print` does not return a useful result when used in an operand expression. On the other hand, we have seen that functions such as `max`, `pow` and `sqrt` can be used effectively in nested expressions.

Second, pure functions tend to be simpler to test. A list of arguments will always lead to the same return value, which can be compared to the expected return value. Testing is discussed in more detail later in this chapter.

For these reasons, we concentrate heavily on creating and using pure functions in the remainder of this chapter. The `print` function is only used so that we can see the intermediate results of computations.

1.8 Types

In Python every value has a type; `type` is a special built-in function that yields the type of a value:

```
>>> type(42)
<class 'int'>
>>> type(-1)
<class 'int'>
>>> type(3.5)
<class 'float'>
```

We say that `42` has type `int` (integer) and `3.5` has type `float` (floating point). So Python actually has two different types of numbers. `42` and `42.0` are actually different values of different types.

Values of type `int` are integers of any size (as long as they fit into the computer memory), whereas values of type `float` are approximations of real (rational and irrational) numbers. Internally a binary representation is used. Therefore unexpected rounding errors occur easily, for example

```
>>> 0.1 + 0.2
0.30000000000000004
```

Hence `float`s should always be used with some caution and never where exact results are required.

Python allows you to combine values of both types with numerical operators

```
>>> 42 + 3.5
45.5
```

but there are exceptions. The function `round` can take one argument and then returns an `int`:

```
>>> round(3.5)
4
>>> type(round(3.5))
<class 'int'>
```

The function `round` can also take as second argument the number of decimal places to round to:

```
>>> round(pi,2)
3.14
>>> round(pi,5)
3.14159
```

However, a `float` as second argument makes no sense and will yield a *type error*:

```
>>> round(pi,2.3)
Traceback (most recent call last):
  File "<python-input-18>", line 1, in <module>
    round(pi,2.3)
    ~~~~~~~~~~~~
TypeError: 'float' object cannot be interpreted as an integer
```

Besides `int` and `float` we already used one other type:

```
>>> type(None)
<class 'NoneType'>
```

Python is a *dynamically typed* programming language. That means that every value (object) has a type and primitive functions always check that their arguments are of expected type. If they are not, then evaluation will be aborted with a type error message.

In contrast, many other programming languages such as Java, C and C++ are *statically typed*. That means that before a program is run, the compiler will check that it is impossible for the program to ever cause a type error at runtime. This check that certain kinds of errors will never occur at runtime helps writing correct programs, but it does make a language slightly more complicated. Statically typed programming languages require some type annotations in a program. Recent versions of Python allow optional type annotations as well, which can be checked with a separate tool; we will not explore this feature here.

Python is a *strongly typed* programming language. Every value can only be used consistently with its type, that is, the actual binary representation of a value is never misinterpreted as a value of a different type, which could cause arbitrary behaviour of the computation. Many older programming languages such as C and C++ are not strongly typed, but Python and for example Java are.

1.9 Strings

Python can manipulate text (values of type `str`, so-called "strings") as well as numbers. This includes characters "!", words "rabbit", sentences "Got your back.", etc. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result. We can do some calculations with operators `+` and `*`:

```
>>> type('hello')
<class 'str'>
>>> type("world")
<class 'str'>
>>> 'hello' + "world"
'helloworld'
>>> 3 * "ha"
'hahaha'
```

You can access individual characters in a string, using their position (index). Note that the first character is at index 0, the second at index 1, etc. Negative indices count from the end. A single character is still a string.

```
>>> name = 'Monty Python'
>>> name
'Monty Python'
>>> name[0]
'M'
>>> name[5]
' '
>>> name[-2]
'o'
>>> name[20]
Traceback (most recent call last):
  File "<python-input-33>", line 1, in <module>
    name[20]
    ~~~~^^^^
IndexError: string index out of range
>>> type(name[0])
<class 'str'>
```

There is a function to determine the length of a string:

```
>>> len(name)
12
```

```
>>> type(len('hello'))
<class 'int'>
```

Slicing is a powerful way to compute substrings using a special syntax. `string[start:stop]` yields the substring starting at index `start` (inclusive) and ending just before index `stop` (exclusive). `string[start:stop:step]` additionally states how many characters to skip. The arguments `start` and `stop` can be omitted, then defaulting to the beginning and end of the given string. For example:

```
>>> name = 'Monty Python'
>>> name[3:7]
'ty P'
>>> name[:7]
'Monty P'
>>> name[2:9:2]
'nyPt'
>>> name[::3]
'MtPh'
>>> name[::-1]
'nohtyP ytnoM'
```

So we can use Python as a calculator not just for numbers, but also for strings.

1.10 From Calculator to Programs

The Python interpreter provides a REPL, a read-evaluate-print-loop. Like a calculator it reads in an expression, evaluates it, prints its value and then repeats from the start. The REPL is useful for exploration and testing, but not for writing many lines of code.

Python code — called a *program* — can be written in a text file, which by convention has the extension `.py`. The Python interpreter can then be asked to execute all lines of the file, one after the other.

Instead of writing in the REPL

```
>>> from math import pi
>>> radius = 11
>>> area = pi * radius * radius
>>> area
380.1327110843649
```

we write the program

```
from math import pi
radius = 11
area = pi * radius * radius
print(area)
```

in a text file `test.py` and call the Python interpreter with it, yielding the output

```
380.1327110843649
```

The file can be written with any text editor (but not a word processor such as Word, which includes formatting information for e.g. bold and underlined text). It is more convenient to use an IDE (integrated development environment). Using professional IDEs such as VSCode and PyCharm requires some learning. The IDE IDLE targets beginners and is included in the Python distribution (so you already have it). IDLE starts with a window that provides the familiar Python interpreter. The menu item `File -> New File` opens a new window for editing a Python program. The editor provides syntax highlighting, that is, different language constructs are shown in different colours to support reading. The menu item `Run -> Run Module` saves the edited program (confirming with you) and then loads it into the interpreter. Afterwards you can use in the REPL all the names bound by the program. The program itself can easily be read, modified and saved for future use.

2 Defining New Functions

We have identified in Python some of the elements that must appear in any powerful programming language:

1. Numbers and arithmetic operations are *primitive* built-in data values and functions.
2. Nested function application provides a means of *combining* operations.
3. Binding names to values provides a limited means of *abstraction*.

2.1 Function Definitions

Now we will learn about function definitions, a much more powerful abstraction technique by which a name can be bound to a compound operation, which can then be referred to as a unit.

We begin by examining how to express the idea of squaring. We might say, "To square something, multiply it by itself." This is expressed in Python as

```
>>> def square(x):
        return mul(x, x)
```

which defines a new function that has been given the name `square`. This user-defined function is not built into the interpreter. It represents the compound operation of multiplying something by itself. The `x` in this definition is called a *formal parameter*, which provides a name for the thing to be multiplied. The definition creates this user-defined function and associates it with the name `square`.

How to define a function. Function definitions consist of a `def` statement that indicates a <name> and a comma-separated list of named <formal parameters>, then a return statement, called the function body, that specifies the <return expression> of the function, which is an expression to be evaluated whenever the function is applied:

```
def <name>(<formal parameters>):
    return <return expression>
```

The second line must be indented — most programmers use four spaces to indent. The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied.

Having defined square, we can apply it with a call expression:

```
>>> square(21)
441
>>> square(add(2, 5))
49
>>> square(square(3))
81
```

We can also use square as a building block in defining other functions. For example, we can easily define a function `sum_squares` that, given any two numbers as arguments, returns the sum of their squares:

```
>>> def sum_squares(x, y):
        return add(square(x), square(y))
>>> sum_squares(3, 4)
25
```

User-defined functions are used in exactly the same way as built-in functions. Indeed, one cannot tell from the definition of `sum_squares` whether `square` is built into the interpreter, imported from a module, or defined by the user.

Both `def` statements and assignment statements bind names to values, and any existing bindings are lost. For example, g below first refers to a function of no arguments, then a number, and then a different function of two arguments.

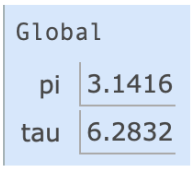
```
>>> def g():
        return 1
>>> g()
1
>>> g = 2
>>> g
2
>>> def g(h, i):
        return h + i
>>> g(1, 2)
3
```

2.2 Environments

Our subset of Python is now complex enough that the meaning of programs is non-obvious. What if a formal parameter has the same name as a built-in function? Can two functions share names without confusion? To resolve such questions, we must describe environments in more detail.

An environment in which an expression is evaluated consists of a sequence of frames, depicted as boxes. Each frame contains bindings, each of which associates a name with its corresponding value. There is a single global frame. Assignment and import statements add entries to the first frame of the current environment. So far, our environment consists only of the global frame.

```
1 from math import pi
2 tau = 2 * pi
```



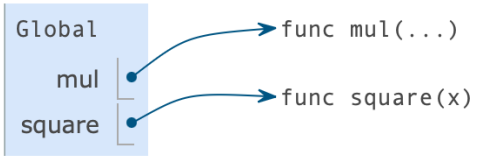
This *environment diagram* shows the bindings of the current environment, along with the values to which names are bound. In general the diagram shows the state of the environment after executing the line indicated by the green triangle (or arrow); the line indicated by a red triangle (or arrow) will be executed next. Above is no red arrow, because execution of the whole program finished. All environment diagrams in this text were generated with the Online Python Tutor[14]. This online tool is interactive: you can write a short Python program and then step through the lines of the program on the left

[14] https://pythontutor.com/cp/composingprograms.html

to see the state of the environment evolve on the right. You are encouraged to create examples yourself and study the resulting environment diagrams.

Functions appear in environment diagrams as well. An import statement binds a name to a built-in function. A `def` statement binds a name to a user-defined function created by the definition. The resulting environment after importing `mul` and defining `square` appears below:

```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
```

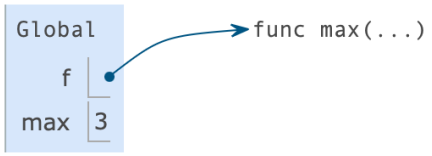


Each function is a line that starts with `func`, followed by the function name and formal parameters. Built-in functions such as `mul` do not have formal parameter names, and so `...` is always used instead.

The name of a function is repeated twice, once in the frame and again as part of the function itself. The name appearing in the function is called the intrinsic name. The name in a frame is a bound name. There is a difference between the two: different names may refer to the same function, but that function itself has only one intrinsic name.

The name bound to a function in a frame is the one used during evaluation. The intrinsic name of a function does not play a role in evaluation. Step through the example below using the Forward button to see that once the name max is bound to the value 3, it can no longer be used as a function.

```
1  f = max
2  max = 3
3  result = f(2, 3, 4)
4  max(1, 2)  # Causes an error
```



The error message `TypeError: 'int' object is not callable` is reporting that the name max (currently bound to the number 3) is an integer and not a function. Therefore, it cannot be used as the operator in a call expression.

Function Signatures. Functions differ in the number of arguments that they are allowed to take. To track these requirements, we draw each function in a way that shows the function name and its formal parameters. The user-defined function `square` takes only `x`; providing more or fewer arguments will result in an error. A description of the formal parameters of a function is called the function's signature.

The function `max` can take an arbitrary number of arguments. It is rendered as `max(...)`. Regardless of the number of arguments taken, all built-in functions will be rendered as `<name>(...)`, because these primitive functions were never explicitly defined.

2.3 Calling User-Defined Functions

To evaluate a call expression whose operator names a user-defined function, the Python interpreter follows a computational process. As with any call expression, the interpreter evaluates the operator and operand expressions, and then applies the named function to the resulting arguments.

Applying a user-defined function introduces a second local frame, which is only accessible to that function. To apply a user-defined function to some arguments:

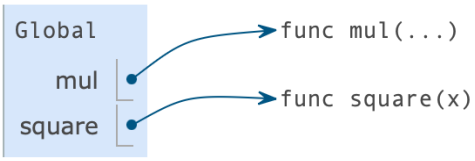
Bind the arguments to the names of the function's formal parameters in a new local frame. Execute the body of the function in the environment that starts with this frame. The environment in which the body is evaluated consists of two frames: first the local frame that contains formal parameter bindings, then the global frame that contains everything else. Each instance of a function application has its own independent local frame.

To illustrate an example in detail, several steps of the environment diagram for the same example are depicted below. After executing the first import statement, only the name `mul` is bound in the global frame.

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

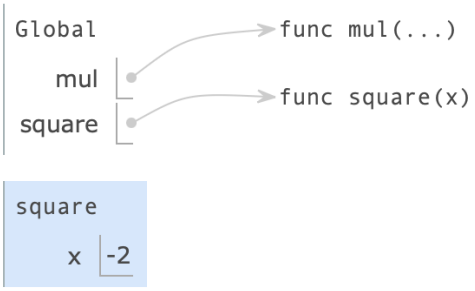
First, the definition statement for the function `square` is executed. Notice that the entire `def` statement is processed in a single step. The body of a function is not executed until the function is called (not when it is defined).

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



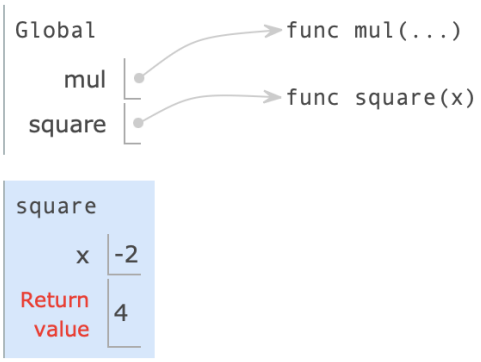
Next, The `square` function is called with the argument `-2`, and so a new frame is created with the formal parameter `x` bound to the value `-2`.

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



Then, the name `x` is looked up in the current environment, which consists of the two frames shown. In both occurrences, `x` evaluates to `-2`, and so the square function returns `4`.

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



The "Return value" in the `square` frame is not a name binding; instead it indicates the value returned by the function call that created the frame.

Even in this simple example, two different environments are used. The top-level expression `square(-2)` is evaluated in the global environment, while the return expression `mul(x, x)` is evaluated in the environment created for by calling `square`. Both `x` and `mul` are bound in this environment, but in different frames.

The order of frames in an environment affects the value returned by looking up a name in an expression. We stated previously that a name is evaluated to the value associated with that name in the current environment. We can now be more precise:

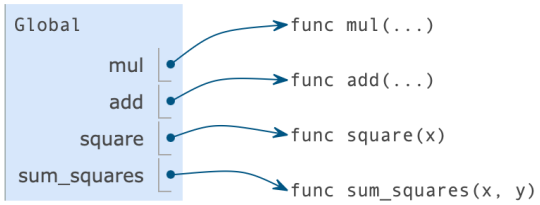
Name Evaluation. A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Evaluation Model. Our conceptual framework of environments, names, and functions constitutes a model of evaluation; while some mechanical details are still unspecified (e.g., how a binding is implemented), our model does precisely and correctly describe how the interpreter evaluates call expressions.

2.4 Example: Calling a User-Defined Function

Let us again consider our two simple function definitions and illustrate the process that evaluates a call expression for a user-defined function.

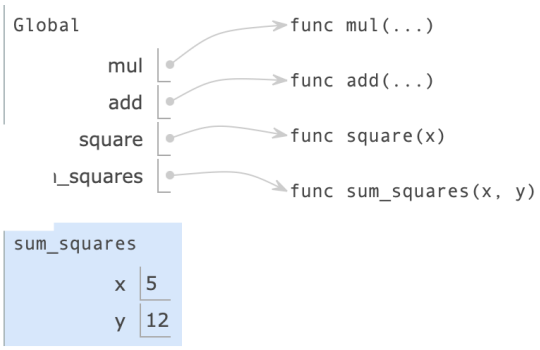
```
1 from operator import add, mul
2 def square(x):
3     return mul(x, x)
4
5 def sum_squares(x, y):
6     return add(square(x), square(y))
7
8 result = sum_squares(5, 12)
```



Python first evaluates the name `sum_squares`, which is bound to a user-defined function in the global frame. The primitive numeric expressions `5` and `12` evaluate to the numbers they represent.

Next, Python applies `sum_squares`, which introduces a local frame that binds `x` to `5` and `y` to `12`.

```
1 from operator import add, mul
2 def square(x):
3     return mul(x, x)
4
5 def sum_squares(x, y):
6     return add(square(x), square(y
7
8 result = sum_squares(5, 12)
```



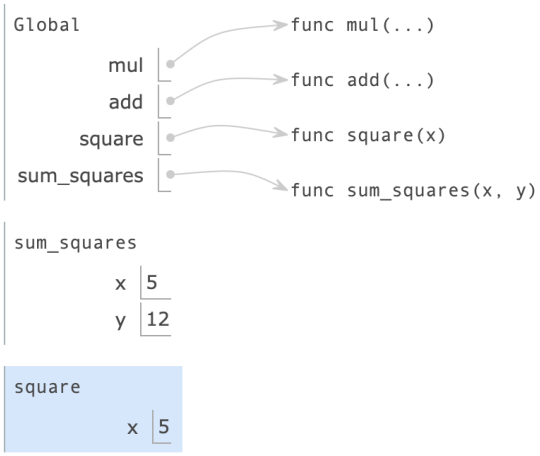
The body of `sum_squares` contains this call expression:

```
  add     (  square(x)   ,   square(y)   )
--------     ---------      ---------
operator     operand 0      operand 1
```

All three subexpressions are evaluated in the current environment, which begins with the frame labeled `sum_squares`. The operator subexpression `add` is a name found in the global frame, bound to the built-in function for addition. The two operand subexpressions must be evaluated in turn, before addition is applied. Both operands are evaluated in the current environment beginning with the frame labeled `sum_squares`.

In operand 0, `square` names a user-defined function in the global frame, while `x` names the number `5` in the local frame. Python applies `square` to `5` by introducing yet another local frame that binds `x` to `5`.

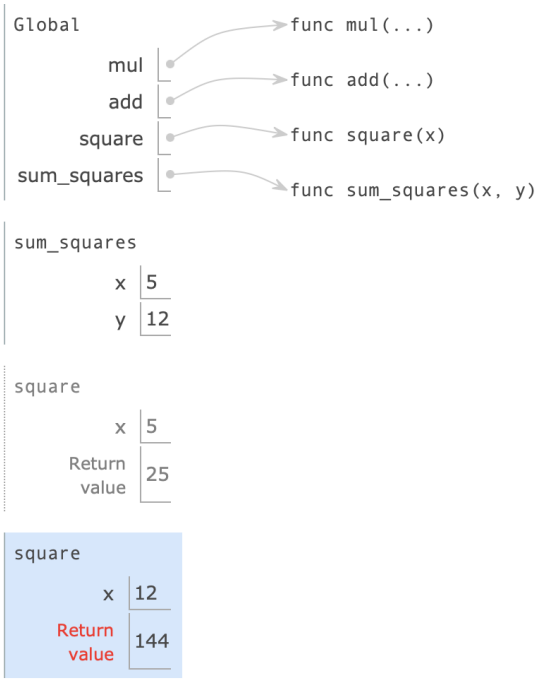
```
1 from operator import add, mul
2 def square(x):
3     return mul(x, x)
4
5 def sum_squares(x, y):
6     return add(square(x), square(y))
7
8 result = sum_squares(5, 12)
```



Using this environment, the expression `mul(x, x)` evaluates to 25.

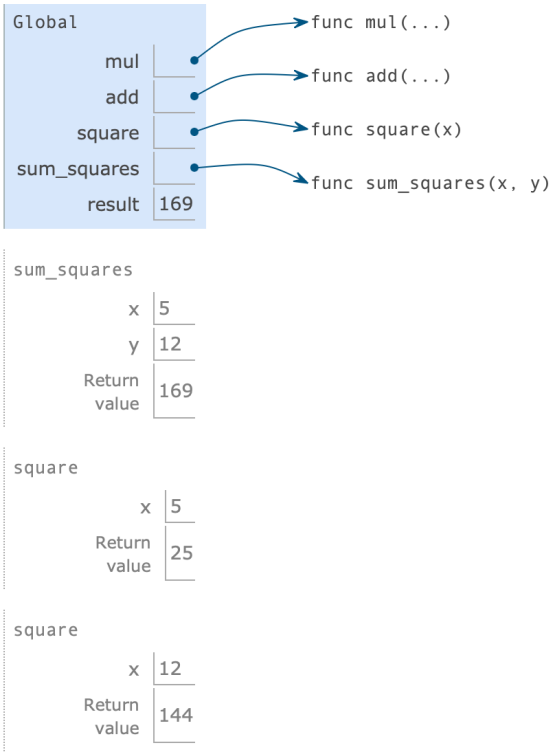
Our evaluation procedure now turns to operand 1, for which `y` names the number 12. Python evaluates the body of `square` again, this time introducing yet another local frame that binds `x` to 12. Hence, operand 1 evaluates to 144.

```
1 from operator import add, mul
2 def square(x):
3     return mul(x, x)
4
5 def sum_squares(x, y):
6     return add(square(x), square(y))
7
8 result = sum_squares(5, 12)
```



Finally, applying addition to the arguments 25 and 144 yields a final return value for `sum_squares`: 169.

```
1 from operator import add, mul
2 def square(x):
3     return mul(x, x)
4
5 def sum_squares(x, y):
6     return add(square(x), square(y))
7
8 result = sum_squares(5, 12)
```



This example illustrates many of the fundamental ideas we have developed so far. Names are bound to values, which are distributed across many independent local frames, along with a single global frame that contains shared names. A new local frame is introduced every time a function is called, even if the same function is called twice.

All of this machinery exists to ensure that names resolve to the correct values at the correct times during program execution. This example illustrates why our model requires the complexity that we have introduced. All three local frames contain a binding for the name `x`, but that name is bound to different values in different frames. Local frames keep these names separate.

2.5 Local Names

One detail of a function's implementation that should not affect the function's behavior is the implementer's choice of names for the function's formal parameters. Thus, the following functions should provide the same behavior:

```
>>> def square(x):
        return mul(x, x)
>>> def square(y):
        return mul(y, y)
```

This principle — that the meaning of a function should be independent of the parameter names chosen by its author — has important consequences for programming languages. The simplest consequence is that the parameter names of a function must remain local to the body of the function.

If the parameters were not local to the bodies of their respective functions, then the parameter `x` in `square` could be confused with the parameter `x` in `sum_squares`. Critically, this is not the case: the binding for `x` in different local frames are unrelated. The model of computation is carefully designed to ensure this independence.

We say that the scope of a local name is limited to the body of the user-defined function that defines it. When a name is no longer accessible, it is out of scope. This scoping behavior isn't a new fact about our model; it is a consequence of the way environments work.

2.6 Choosing Names

The interchangeability of names does not imply that formal parameter names do not matter at all. On the contrary, well-chosen function and parameter names are essential for the human interpretability of function definitions!

The following guidelines are adapted from the style guide for Python code, which serves as a guide for all (non-rebellious) Python programmers. A shared set of conventions smooths communication among members of a developer community. As a side effect of following these conventions, you will find that your code becomes more internally consistent.

1. Function names are lowercase, with words separated by underscores. Descriptive names are encouraged.
2. Function names typically evoke operations applied to arguments by the interpreter (e.g., print, add, square) or the name of the quantity that results (e.g., max, abs, sum).
3. Parameter names are lowercase, with words separated by underscores. Single-word names are preferred.
4. Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed.
5. Single letter parameter names are acceptable when their role is obvious, but avoid `"l"` (lowercase ell), `"O"` (capital oh), or `"I"` (capital i) to avoid confusion with numerals.

There are many exceptions to these guidelines, even in the Python standard library. Like the vocabulary of the English language, Python has inherited words from a variety of contributors, and the result is not always consistent.

2.7 Functions as Abstractions

Though it is very simple, `sum_squares` exemplifies the most powerful property of user-defined functions. The function `sum_squares` is defined in terms of the function `square`, but relies only on the relationship that `square` defines between its input arguments and its output values.

We can write `sum_squares` without concerning ourselves with how to square a number. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as `sum_squares` is concerned, `square` is not a particular function body, but rather an abstraction of a function, a so-called functional abstraction. At this level of abstraction, any function that computes the square is equally good.

Thus, considering only the values they return, the following two functions for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.

```
>>> def square(x):
        return mul(x, x)
>>> def square(x):
        return mul(x, x-1) + x
```

In other words, a function definition should be able to suppress details. The users of the function may not have written the function themselves, but may have obtained it from another programmer as a "black box". A programmer should not need to know how the function is implemented in order to use it. The Python Library has this property. Many developers use the functions defined there, but few ever inspect their implementation.

2.8 Operators

Mathematical operators (such as `+` and `-`) provided our first example of a method of combination, but we have yet to define an evaluation procedure for expressions that contain these operators.

Python expressions with infix operators each have their own evaluation procedures, but you can often think of them as short-hand for call expressions. When you see

```
>>> 2 + 3
5
```

simply consider it to be shorthand for

```
>>> add(2, 3)
5
```

Infix notation can be nested, just like call expressions. Python applies the normal mathematical rules of *operator precedence*, which dictate how to interpret a compound expression with multiple operators.

```
>>> 2 + 3 * 4 + 5
19
```

evaluates to the same result as

```
>>> add(add(2, mul(3, 4)), 5)
19
```

The nesting in the call expression is more explicit than the operator version, but also harder to read. Python also allows subexpression grouping with parentheses, to override the normal precedence rules or make the nested structure of an expression more explicit.

```
>>> (2 + 3) * (4 + 5)
45
```

evaluates to the same result as

```
>>> mul(add(2, 3), add(4, 5))
45
```

When it comes to division, Python provides two infix operators: `/` and `//`. The former is normal division, so that it results in a floating point, or decimal value, even if the divisor evenly divides the dividend:

```
>>> 5 / 4
1.25
>>> 8 / 4
2.0
```

The `//` operator, on the other hand, rounds the result down to an integer:

```
>>> 5 // 4
1
>>> -5 // 4
-2
```

These two operators are shorthand for the `truediv` and `floordiv` functions.

```
>>> from operator import truediv, floordiv
>>> truediv(5, 4)
1.25
>>> floordiv(5, 4)
1
```

You should feel free to use infix operators and parentheses in your programs. Idiomatic Python prefers operators over call expressions for simple mathematical operations.

Just like the `operator` module has functions for the operators `+`, `-`, etc., it also contains functions for string indexing and slicing. So even though Python does have special evaluation procedures for indexing and slicing, we can just see the special syntax such as `string[index]` and `string[start:stop:step]` as a shorthand for using the corresponding operators in call expressions.

2.9 Objects and Method Calls

There is one further special form of expression that you may have already seen in some Python programs:

```
>>> text = 'Monty Python'
>>> text.upper()
'MONTY PYTHON'
>>> text.lower()
'monty python'
>>> text.replace('Python','Don')
'Monty Don'
>>> text.find('th')
8
>>> text.replace('Python','Don').upper()
'MONTY DON'
```

The names `upper`, `lower`, `replace`, etc. denote *methods*, which are special functions bound to an object of type `str`.

$$expr.name(arg_1, \ldots, arg_n)$$

To evaluate such a *method call expression*, the interpreter first evaluates the *expr* and the arguments arg_1 to arg_n. The *expr* evaluates to an object. Every object has its own environment, based on its type, in which the *name* is looked up, yielding a special function. That function is finally applied to the object (as first argument) and all the values of arg_1 to arg_n.

So a *method* is a special function that is bound in the environment of an object and that will always get that object as first argument. Evaluation of a method call expression is more complicated than evaluation of a (function) call expression, but not fundamentally different.

Note that the Online Python Tutor[15] does not display the environments of objects at all, even though it can evaluate method call expressions.

We will later explore the benefits of each object having its own environment of methods. For the moment let us just note that while this increases the number of environments, it means that in a large program we can avoid having a high number of names bound in a single environment; now the same name can be used in different environments for different methods.

There are very few methods for objects of the numeric types `int` and `float`; operators and functions are preferred.

```
>>> (4243134).bit_length()
23
```

We will later see other types that have objects with many methods.

3 Control

The expressive power of the functions that we can define at this point is very limited, because we have not introduced a way to make comparisons and to perform different operations depending on the result of a comparison. Control statements will give us this ability. They are statements that control the flow of a program's execution based on the results of logical comparisons.

Statements differ fundamentally from the expressions that we have studied so far. They have no value. Instead of computing something, executing a control statement determines what the interpreter should do next.

[15] https://pythontutor.com/cp/composingprograms.html

3.1 Statements

So far, we have primarily considered how to evaluate expressions. However, we have seen three kinds of statements already: assignment, def, and return statements. These lines of Python code are not themselves expressions, although they all contain expressions as components.

Rather than being evaluated, statements are executed. Each statement describes some change to the interpreter state, and executing a statement applies that change. As we have seen for return and assignment statements, executing statements can involve evaluating subexpressions contained within them.

Expressions can also be executed as statements, in which case they are evaluated, but their value is discarded. Executing a pure function has no effect, but executing a non-pure function can cause effects as a consequence of function application.

Consider, for instance,

```
>>> def square(x):
        mul(x, x) # Watch out! This call doesn't return a value.
```

This example is valid Python, but probably not what was intended. The body of the function consists of an expression. An expression by itself is a valid statement, but the effect of the statement is that the `mul` function is called, and the result is discarded. If you want to do something with the result of an expression, you need to say so: you might store it with an assignment statement or return it with a return statement:

```
>>> def square(x):
        return mul(x, x)
```

Sometimes it does make sense to have a function whose body is an expression, when a non-pure function like `print` is called.

```
>>> def print_square(x):
        print(square(x))
```

At its highest level, the Python interpreter's job is to execute programs, composed of statements. However, much of the interesting work of computation comes from evaluating expressions. Statements govern the relationship among different expressions in a program and what happens to their results.

3.2 Compound Statements

In general, Python code is a sequence of statements. A simple statement is a single line that doesn't end in a colon. A compound statement is so called because it is composed of other statements (simple and compound). Compound statements typically span multiple lines and start with a one-line *header* ending in a colon, which identifies the type of statement. Together, a header and an indented *suite* of statements is called a *clause*. A *compound statement* consists of one or more clauses:

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

We can understand the statements we have already introduced in these terms.

- Expressions, `return` statements, and assignment statements are simple statements.
- A `def` statement is a compound statement. The suite that follows the `def` header defines the function body.

Specialised evaluation rules for each kind of header dictate when and if the statements in its suite are executed. We say that the header controls its suite. For example, in the case of `def` statements, we saw that the `return` expression is not evaluated immediately, but instead stored for later use when the defined function is eventually called.

We can also understand multi-line programs now.

- To execute a sequence of statements, execute the first statement. If that statement does not redirect control, then proceed to execute the rest of the sequence of statements, if any remain.

This definition exposes the essential structure of a recursively defined sequence: a sequence can be decomposed into its first element and the rest of its elements. The "rest" of a sequence of statements is itself a sequence of statements! Thus, we can recursively apply this execution rule. This view of sequences as recursive data structures will appear again in later chapters.

The important consequence of this rule is that statements are executed in order, but later statements may never be reached, because of redirected control.

Practical Guidance. When indenting a suite, all lines must be indented the same amount and in the same way (use spaces, not tabs). Any variation in indentation will cause an error. Using 4 spaces is standard practice.

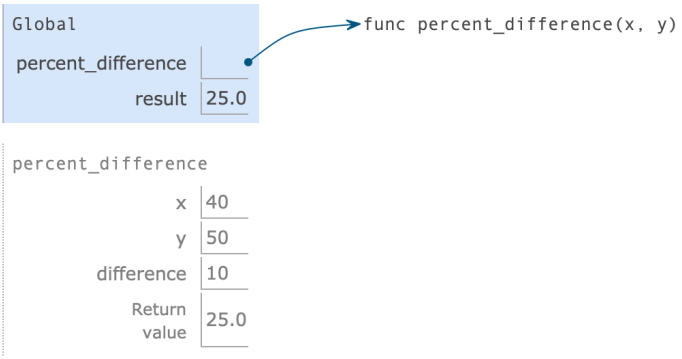
3.3 Defining Functions II: Local Assignment

Originally, we stated that the body of a user-defined function consisted only of a return statement with a single return expression. In fact, functions can define a sequence of operations that extends beyond a single expression.

Whenever a user-defined function is applied, the sequence of clauses in the suite of its definition is executed in a local environment — an environment starting with a local frame created by calling that function. A return statement redirects control: the process of function application terminates whenever the first return statement is executed, and the value of the return expression is the returned value of the function being applied.

Assignment statements can appear within a function body. For instance, this function returns the absolute difference between two quantities as a percentage of the first, using a two-step calculation:

```
1 def percent_difference(x, y):
2     difference = abs(x-y)
3     return 100 * difference / x
4 result = percent_difference(40, 50)
```



The effect of an assignment statement is to bind a name to a value in the first frame of the current environment. As a consequence, assignment statements within a function body cannot affect the global frame. The fact that functions can only manipulate their local environment is critical to creating modular programs, in which pure functions interact only via the values they take and return.

Of course, the `percent_difference` function could be written as a single expression, as shown below, but the return expression is more complex.

```
>>> def percent_difference(x, y):
        return 100 * abs(x-y) / x
>>> percent_difference(40, 50)
25.0
```

So far, local assignment hasn't increased the expressive power of our function definitions. It will do so, when combined with other control statements. In addition, local assignment also plays a critical role in clarifying the meaning of complex expressions by assigning names to intermediate quantities.

3.4 Conditional Statements

Python has a built-in function for computing absolute values.

```
>>> abs(-2)
2
```

We would like to be able to implement such a function ourselves, but we have no obvious way to define a function that has a comparison and a choice. We would like to express that if `x` is positive, `abs(x)` returns `x`. Furthermore, if `x` is `0`, `abs(x)` returns `0`. Otherwise, `abs(x)` returns `-x`. In Python, we can express this choice with a conditional statement.

```
def absolute_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

result = absolute_value(-2)
```

This implementation of `absolute_value` raises several important issues:

Conditional statements. A conditional statement in Python consists of a series of headers and suites: a required `if` clause, an optional sequence of `elif` clauses, and finally an optional `else` clause:

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

When executing a conditional statement, each clause is considered in order. The computational process of executing a conditional clause follows.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite. Then, skip over all subsequent clauses in the conditional statement.

If the `else` clause is reached (which only happens if all `if` and `elif` expressions evaluate to false values), its suite is executed.

Boolean contexts. Above, the execution procedures mention "a false value" and "a true value." The expressions inside the header statements of conditional blocks are said to be in boolean contexts: their truth values matter to control flow, but otherwise their values are not assigned or returned. Python includes several false values, including `0`, `None`, and the boolean value `False`. All other numbers are true values. Every built-in kind of data in Python has both true and false values.

Boolean values. Python has two boolean values, called `True` and `False`. Boolean values represent truth values in logical expressions. The built-in comparison operations, `>`, `<`, `>=`, `<=`, `==`, `!=`, return these values.

```
>>> 4 < 2
False
>>> 5 >= 5
True
```

This second example reads "5 is greater than or equal to 5", and corresponds to the function `ge` in the operator module.

```
>>> 0 == -0
True
```

This final example reads "0 equals -0", and corresponds to `eq` in the operator module. Notice that Python distinguishes assignment (`=`) from equality comparison (`==`), a convention shared across many programming languages.

Boolean operators. Three basic logical operators are also built into Python:

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

Logical expressions have corresponding evaluation procedures. These procedures exploit the fact that the truth value of a logical expression can sometimes be determined without evaluating all of its subexpressions, a feature called short-circuiting.

To evaluate the expression `<left> and <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a false value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `<left> or <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a true value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `not <exp>`:

1. Evaluate `<exp>`; The value is `True` if the result is a false value, and `False` otherwise.

These values, rules, and operators provide us with a way to combine the results of comparisons. Functions that perform comparisons and return boolean values typically begin with `is`, not followed by an underscore (e.g., `isfinite`, `isdigit`, `isinstance`, etc.).

3.5 Iteration

In addition to selecting which statements to execute, control statements are used to express repetition. If each line of code we wrote were only executed once, programming would be a very unproductive exercise. Only through repeated execution of statements do we unlock the full potential of computers. We have already seen one form of repetition: a function can be applied many times, although it is only defined once. Iterative control structures are another mechanism for executing the same statements many times.

Consider the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

`0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

Each value is constructed by repeatedly applying the sum-previous-two rule. The first and second are fixed to 0 and 1. For instance, the eighth Fibonacci number is 13.

We can use a `while` statement to enumerate `n` Fibonacci numbers. We need to track how many values we've created (`k`), along with the `k`th value (`curr`) and its predecessor (`pred`). Step through this function and observe how the Fibonacci numbers evolve one by one, bound to `curr`.

```
def fib(n):
    pred, curr = 0, 1   # Fibonacci numbers 1 and 2
    k = 2               # Which Fib number is curr?
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr

result = fib(8)
```

Remember that commas separate multiple names and values in an assignment statement. The line:

```
pred, curr = curr, pred + curr
```

has the effect of rebinding the name `pred` to the value of `curr`, and simultaneously rebinding `curr` to the value of `pred + curr`. All of the expressions to the right of `=` are evaluated before any rebinding takes place.

This order of events — evaluating everything on the right of `=` before updating any bindings on the left — is essential for correctness of this function.

A `while` clause contains a header expression followed by a suite:

```
while <expression>:
    <suite>
```

To execute a while clause:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.

In step 2, the entire suite of the `while` clause is executed before the header expression is evaluated again.

To prevent the suite of a `while` clause from being executed indefinitely, the suite should always change some binding in each pass.

A `while` statement that does not terminate is called an infinite loop. Press `<Control>-C` to force Python to stop looping.

4 Designing Functions

Functions are an essential ingredient of all programs, large and small, and serve as our primary medium to express computational processes in a programming language. So far, we have discussed the formal properties of functions and how they are applied. We now turn to the topic of what makes a good function. Fundamentally, the qualities of good functions all reinforce the idea that functions are abstractions.

- Each function should have exactly one job. That job should be identifiable with a short name and characterizable in a single line of text. Functions that perform multiple jobs in sequence should be divided into multiple functions.
- *Don't repeat yourself* is a central tenet of software engineering. The so-called *DRY principle* states that multiple fragments of code should not describe redundant logic. Instead, that logic should be implemented once, given a name, and applied multiple times. If you find yourself copying and pasting a block of code, you have probably found an opportunity for functional abstraction.
- Functions should be defined generally. Squaring is not in the Python Library precisely because it is a special case of the `pow` function, which raises numbers to arbitrary powers.

These guidelines improve the readability of code, reduce the number of errors, and often minimize the total amount of code written. Decomposing a complex task into concise functions is a skill that takes experience to master. Fortunately, Python provides several features to support your efforts.

4.1 Documentation

A function definition will often include documentation describing the function, called a *docstring*, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behaviour of the function:

```
def cylinder_volume(r, h):
    """Compute the volume of a cylinder.

    r - radius, in meters
    h - height, in meters
    result in cubic meters
    """
    return pi * r * r * h
```

When you call help with the name of a function as an argument, you see its docstring (type q to quit Python help).

```
>>> help(cylinder_volume)
```

When writing Python programs, include docstrings for all but the simplest functions. Remember, code is written only once, but often read many times. The Python docs include docstring guidelines[16] that maintain consistency across different Python projects.

Comments. Comments in Python can be attached to the end of a line following the `#` symbol. For example, the comment `Boltzmann's constant` above describes `k`. These comments don't ever appear in Python's help, and they are ignored by the interpreter. They exist for humans alone.

4.2 Testing

Testing a function is the act of verifying that the function's behaviour matches expectations. Our language of functions is now sufficiently complex that we need to start testing our implementations.

A test is a mechanism for systematically performing this verification. Tests typically take the form of one or more sample calls to the function being tested. The returned value is then verified against an expected result. Unlike most functions, which are meant to be general, tests involve selecting and validating calls with specific argument values. Tests also serve as documentation: they demonstrate how to call a function and what argument values are appropriate.

Assertions. Programmers use `assert` statements to verify expectations, such as the output of a function being tested. An `assert` statement has an expression in a boolean context.

[16] https://peps.python.org/pep-0257/

```
assert square(7) == 49
```

When the expression being asserted evaluates to a true value, executing an `assert` statement has no effect. When it is a false value, assert causes an error that halts execution.

We should always test several arguments, including extreme values.

```
assert square(0) == 0
assert square(1) == 1
assert square(-2) == 4
assert square(3) == 9
```

When writing Python in files, rather than directly into the interpreter, tests are typically written in the same file or a neighbouring file with the suffix `_test.py`.

The key to effective testing is to write (and run) tests immediately after implementing new functions.

A test that applies a single function is called a *unit test*. Exhaustive unit testing is a hallmark of good program design.

4.3 Development of a Function Definition

It is good practice to write some tests before you implement a function, in order to have some example inputs and outputs in your mind.

So if we want to write a function `cubic` we may first write

```
assert cubic(3) == 27
assert cubic(0) == 0
assert cubic(-2) == 4
```

then write the function definition

```
def cubic(x):
    """Compute the cubic of a given number."""
    return x * x * x
```

and finally add some more tests:

```
import math  # this line should be at the beginning of the program

assert math.isclose(cubic(0.3),0.27)
assert math.isclose(cubic(-0.1),0.001)
```

4.4 Programming by Contract

The docstring of a function documents what the function does. However, the docstring of `cylinder_volume` is not a complete specification. The first line describes the job of the function in a single line. That is necessarily only partial information. What additional information should a docstring provide? What does it mean for a function definition to be correct?

The idea of *programming by contract* is that every function should come with a contract:

Precondition: Restrictions on parameters; conditions that must hold before a function is called.

Postcondition: Promises about the result value and actions taken; conditions that will hold afterwards.

Contract: Precondition + postcondition

The docstring of our earlier definition of `cylinder_volume` states both a precondition and a postcondition. The precondition is that the first formal parameter is the cylinder radius, in meters, and the second formal parameter is the cylinder height, in meters. The postcondition is that the result is the volume of the cylinder in cubic meters.

The statement about parameters being "in meters" and the result "in cubic meters" implicitly means that all these are numbers, that is values of type `int` or `float`, not for example a value of type `str`. Additionally, we note that the function should never be applied to negative arguments: there is no negative radius or height. Note that we have specified the *domain* of the function, the set of arguments it can take, and its *range*, the set of values it can return. Domain and range are usually part of a function's contract.

We should clarify the contract of `cylinder_volume` by expanding the docstring as follows:

```
def cylinder_volume(r, h):
    """Compute the volume of the cylinder.

    r - radius, in meters; number >= 0
    h - height, in meters; number >= 0
    result in cubic meters; number >= 0
    """
    return pi * r * r * h
```

So in this docstring the lines

```
def cylinder_volume(r, h):
```

```
r - radius, in meters; number >= 0
h - height, in meters; number >= 0
```

express the precondition (the first line states the number of formal parameters and names that relate to the latter lines) and the lines

```
"""Compute the volume of the cylinder.

result in cubic meters; number >= 0
```

express the postcondition.

A function definition is *correct* with respect to its contract:

> If the function caller meets the precondition, then the function will meet the postcondition.

Note that if the caller doesn't meet the precondition, then the function can do anything. So a call `cylinder_volume("2m", "3m")` would clearly violate the precondition and hence the function would be free to do anything, including aborting with an error message or returning an arbitrary value. The blame of the failure clearly lies with the caller of `cylinder_volume`, not the definition of the function itself.

Note that many texts on *programming by contract* state that a function should check that its precondition is met by using assertions (and possibly also include assertions for the postcondition). However, for formulating a contract and thus expressing what it means for a function definition to be correct, that is not required. In fact, checking pre- and postconditions with assertions has disadvantages:

- Evaluating an assertion costs extra time; note that assertions for pre- and postconditions would be evaluated during normal program execution, not during prior testing.
- If every pre- and every postcondition is checked, then the same condition will be checked numerous times for the same arguments during program execution.
- The concept of contracts allows expressing in natural language preconditions and postconditions that are difficult or even impossible to check automatically.

A contract does not specify how the function works; that detail is abstracted away.

Another example with contract and tests:

```
def sum_naturals(n):
    """Return the sum of the first n natural numbers.

    n is an int >= 0
    result is an int >= 0
    """
    total = 0
    k = 1
    while k <= n:
        total = total + k
        k = k + 1
    return total

assert sum_naturals(0) == 0
assert sum_naturals(10) == 55
assert sum_naturals(100) == 5050
```

4.5 Debugging

Programmers make mistakes. Most large programs have *defects*. Running a defective program may cause a *failure*, an externally visible error (for example a wrong or missing result). For ensuring that a program is correct, it is useful to carefully distinguish between the defect in a program and the failure that it causes. However, in practice the word *bug* has been established for both. Correcting a defective program is called *debugging*. The most time consuming part of debugging is usually determining from an observed failure the defect in the program that caused it; often people only mean this defect localisation with the word *debugging*.

Testing finds failures, not defects. Testing can be used to show the presence of bugs, but never to show their absence (The Humble Programmer by Edsger W. Dijkstra, 1972[17]).

Nonetheless, systematically writing and running unit tests for every function definitions is substantial help in writing correct programs. Running a large number of unit tests for a function gives us no guarantee, but some confidence that the function definition is correct. Hence, when a unit test of a function fails, we can be quite sure that the defect is in the definition of that function, not another function called from the function, because all unit tests for those functions pass.

So locating a defect in a small function definition may not be so hard if we already know that the defect must be in that definition.

Many systematic debugging methods (defect-localisation methods) exist.

[17] https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

Stepping through an execution with a tool like the Python tutor is one method. It enables us to view the intermediate values of names in the environments. Integrated Development Environments (IDEs) usually include such a stepping debugger.

One of the oldest debugging methods is to include `print` statements in the program for displaying the values of some internal names or expressions of interest. The main challenge is to insert `print` statements that display useful information and the right amount of information. Output can easily become too voluminous to be comprehensible. Also removing `print` statements after the defect has been located and corrected can be laborious. *Logging* is an improvement on the `print` method.

To learn more about debugging see Andreas Zeller's debugging book[18].

4.6 Extra: Default Argument Values

A consequence of defining general functions is the introduction of additional arguments. Functions with many arguments can be awkward to call and difficult to read.

In Python, we can provide default values for the arguments of a function. When calling that function, arguments with default values are optional. If they are not provided, then the default value is bound to the formal parameter name instead. For instance, if an application commonly computes pressure for one mole of particles, this value can be provided as a default:

```
def pressure(v, t, n=6.022e23):
    """Compute the pressure in pascals of an ideal gas.

    v -- volume of gas, in cubic meters
    t -- absolute temperature in degrees kelvin
    n -- particles of gas (default: one mole)
    """
    k = 1.38e-23  # Boltzmann's constant
    return n * k * t / v
```

The `=` symbol means two different things in this example, depending on the context in which it is used. In the `def` statement header, `=` does not perform assignment, but instead indicates a default value to use when the `pressure` function is called. By contrast, the assignment statement to `k` in the body of the function binds the name `k` to an approximation of Boltzmann's constant.

```
>>> pressure(1, 273.15)
2269.974834
```

[18] https://www.debuggingbook.org

```
>>> pressure(1, 273.15, 3 * 6.022e23)
6809.924502
```

The `pressure` function is defined to take three arguments, but only two are provided in the first call expression above. In this case, the value for `n` is taken from the `def` statement default. If a third argument is provided, the default is ignored.

As a guideline, most data values used in a function's body should be expressed as default values to named arguments, so that they are easy to inspect and can be changed by the function caller. Some values that never change, such as the fundamental constant `k`, can be bound in the function body or in the global frame.

5 Higher-Order Functions

We have seen that functions are a method of abstraction that describe compound operations independent of the particular values of their arguments. That is, in square,

```
def square(x):
    return x * x
```

we are not talking about the square of a particular number, but rather about a method for obtaining the square of any number. Of course, we could get along without ever defining this function, by always writing expressions such as

```
>>> 3 * 3
9
>>> 5 * 5
25
```

and never mentioning `square` explicitly. This practice would suffice for simple computations such as `square`, but would become arduous for more complex examples such as `abs` or `fib`. In general, lacking function definition would put us at the disadvantage of forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations. Our programs would be able to compute squares, but our language would lack the ability to express the concept of squaring.

One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the names directly. Functions provide this ability. As we will see in the following examples, there are common programming patterns that recur in code, but are used

with a number of different functions. These patterns can also be abstracted, by giving them names.

To express certain general patterns as named concepts, we will need to construct functions that can accept other functions as arguments or return functions as values. Functions that manipulate functions are called *higher-order functions*. This section shows how higher-order functions can serve as powerful abstraction mechanisms, vastly increasing the expressive power of our language.

5.1 Functions as Arguments

Consider the following three functions, which all compute summations.

```
def sum_naturals(n):
    """Return sum of natural numbers up to n.

    n is an int >= 0
    result is an int >= 0
    """
    total = 0
    k = 1
    while k <= n:
        total = total + k
        k = k + 1
    return total

assert sum_naturals(100) == 5050

def sum_cubes(n):
    """Return sum of the cubes of natural numbers up to n.

    n is an int >= 0
    result is an int >= 0
    """
    total = 0
    k = 1
    while k <= n:
        total = total + k*k*k
        k = k + 1
    return total

assert sum_cubes(100) == 25502500
```

```
def pi_sum(n):
    """Return approximation of pi as sum of converging series terms up to n.

    n is an int >= 0
    result is a float >= 0
    """
    total = 0
    k = 1
    while k <= n:
        total = total + 8 / ((4*k-3) * (4*k-1))
        k = k + 1
    return total

assert math.isclose(pi_sum(100), 3.1365926848388144)
```

The third function, `pi_sum`, computes the sum of terms in the series

$$\frac{8}{1 \cdot 3} + \frac{8}{5 \cdot 7} + \frac{8}{9 \cdot 11} + \ldots$$

which converges to π very slowly.

These three functions clearly share a common underlying pattern. They are for the most part identical, differing only in name and the function of `k` used to compute the term to be added. We could generate each of the functions by filling in slots in the same template:

```
def <name>(n):
    total = 0
    k = 1
    while k <= n:
        total = total + <term>(k)
        k = k + 1
    return total
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Each of these functions is a summation of terms. As program designers, we would like our language to be powerful enough so that we can write a function that expresses the concept of summation itself rather than only functions that compute particular sums. We can do so readily in Python by taking the common template shown above and transforming the "slots" into formal parameters:

In the example below, summation takes as its two arguments the upper bound `n` together with the function `term` that computes the kth term. We can use summation just as we

would any function, and it expresses summations succinctly. Take the time to step through this example, and notice how binding `cube` to the local name `term` ensures that the result 1*1*1 + 2*2*2 + 3*3*3 = 36 is computed correctly.

```
def summation(n, term):
    total = 0
    k = 1
    while k <= n:
        total = total + term(k)
        k = k + 1
    return total

def cube(x):
    return x*x*x

def sum_cubes(n):
    return summation(n, cube)

result = sum_cubes(3)
```

Using an identity function that returns its argument, we can also sum natural numbers using exactly the same summation function.

```
def summation(n, term):
    total = 0
    k = 1
    while k <= n:
       total= total + term(k)
       k = k + 1
    return total
def identity(x):
    return x
def sum_naturals(n):
    return summation(n, identity)
```

```
>>> sum_naturals(10)
55
```

The summation function can also be called directly, without defining another function for a specific sequence.

```
>>> summation(10, square)
385
```

We can define `pi_sum` using our summation abstraction by defining a function `pi_term` to compute each term. We pass the argument `1e6`, a shorthand for `1 * 10^6 = 1000000`, to generate a close approximation to pi.

```
def pi_term(x):
    return 8 / ((4*x-3) * (4*x-1))
def pi_sum(n):
    return summation(n, pi_term)
```

```
>>> pi_sum(1e6)
3.141592153589902
```

5.2 Functions as General Methods

We introduced user-defined functions as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order functions, we begin to see a more powerful kind of abstraction: some functions express general methods of computation, independent of the particular functions they call.

Despite this conceptual extension of what a function means, our environment model of how to evaluate a call expression extends gracefully to the case of higher-order functions, without change. When a user-defined function is applied to some arguments, the formal parameters are bound to the values of those arguments (which may be functions) in a new local frame.

Consider the following example, which implements a general method for iterative improvement and uses it to compute the golden ratio[19]. The golden ratio, often called "phi", is a number near 1.6 that appears frequently in nature, art, and architecture.

An iterative improvement algorithm begins with a guess of a solution to an equation. It repeatedly applies an update function to improve that guess, and applies a close comparison to check whether the current guess is "close enough" to be considered correct.

```
def improve(update, close, guess=1):
    while not close(guess):
        guess = update(guess)
    return guess
```

[19] https://en.wikipedia.org/wiki/Golden_ratio

This `improve` function is a general expression of repetitive refinement. It doesn't specify what problem is being solved: those details are left to the `update` and `close` functions passed in as arguments.

Among the well-known properties of the golden ratio are that it can be computed by repeatedly summing the inverse of any positive number with 1, and that it is one less than its square. We can express these properties as functions to be used with improve.

```
def golden_update(guess):
    return 1/guess + 1
def square_close_to_successor(guess):
    return approx_eq(guess * guess, guess + 1)
```

Above, we introduce a call to `approx_eq` that is meant to return `True` if its arguments are approximately equal to each other. To implement, `approx_eq`, we can compare the absolute value of the difference between two numbers to a small tolerance value.

```
def approx_eq(x, y, tolerance=1e-15):
    return abs(x - y) < tolerance
```

Calling `improve` with the arguments `golden_update` and `square_close_to_successor` will compute a finite approximation to the golden ratio.

```
>>> improve(golden_update, square_close_to_successor)
1.6180339887498951
```

By tracing through the steps of evaluation, we can see how this result is computed. First, a local frame for `improve` is constructed with bindings for `update`, `close`, and `guess`. In the body of `improve`, the name `close` is bound to `square_close_to_successor`, which is called on the initial value of `guess`. Trace through the rest of the steps to see the computational process that evolves to compute the golden ratio.

This example illustrates two related big ideas in computer science. First, naming and functions allow us to abstract away a vast amount of complexity. While each function definition has been trivial, the computational process set in motion by our evaluation procedure is quite intricate. Second, it is only by virtue of the fact that we have an extremely general evaluation procedure for the Python language that small components can be composed into complex processes. Understanding the procedure of interpreting programs allows us to validate and inspect the process we have created.

As always, our new general method `improve` needs a test to check its correctness. The golden ratio can provide such a test, because it also has an exact closed-form solution, which we can compare to this iterative result.

```
from math import sqrt
phi = 1/2 + sqrt(5)/2
approx_phi = improve(golden_update, square_close_to_successor)

assert approx_eq(phi, approx_phi)
```

For this test, no news is good news: the assert statement returns `None` after it is executed successfully.

5.3 Defining Functions III: Nested Definitions

The above examples demonstrate how the ability to pass functions as arguments significantly enhances the expressive power of our programming language. Each general concept or equation maps onto its own short function. One negative consequence of this approach is that the global frame becomes cluttered with names of small functions, which must all be unique. Another problem is that we are constrained by particular function signatures: the `update` argument to `improve` must take exactly one argument. Nested function definitions address both of these problems, but require us to enrich our environment model.

Let's consider a new problem: computing the square root of a number. In programming languages, "square root" is often abbreviated as `sqrt`. Repeated application of the following update converges to the square root of a:

```
def average(x, y):
    return (x + y)/2
def sqrt_update(x, a):
    return average(x, a/x)
```

This two-argument update function is incompatible with `improve` (it takes two arguments, not one), and it provides only a single update, while we really care about taking square roots by repeated updates. The solution to both of these issues is to place function definitions inside the body of other definitions.

```
def sqrt(a):
    def sqrt_update(x):
        return average(x, a/x)
    def sqrt_close(x):
        return approx_eq(x * x, a)
    return improve(sqrt_update, sqrt_close)
```

Like local assignment, local `def` statements only affect the current local frame. These functions are only in scope while `sqrt` is being evaluated. Consistent with our evaluation procedure, these local `def` statements don't even get evaluated until `sqrt` is called.

Lexical scope. Locally defined functions also have access to the name bindings in the scope in which they are defined. In this example, `sqrt_update` refers to the name `a`, which is a formal parameter of its enclosing function `sqrt`. This discipline of sharing names among nested definitions is called *lexical scoping*. Critically, the inner functions have access to the names in the environment where they are defined (not where they are called).

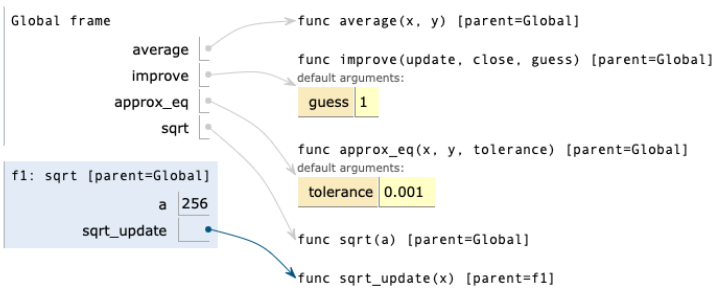
We require two extensions to our environment model to enable lexical scoping.

Each user-defined function has a parent environment: the environment in which it was defined. When a user-defined function is called, its local frame extends its parent environment. Previous to `sqrt`, all functions were defined in the global environment, and so they all had the same parent: the global environment. By contrast, when Python evaluates the first two clauses of `sqrt`, it create functions that are associated with a local environment. In the call

```
>>> sqrt(256)
16.0
```

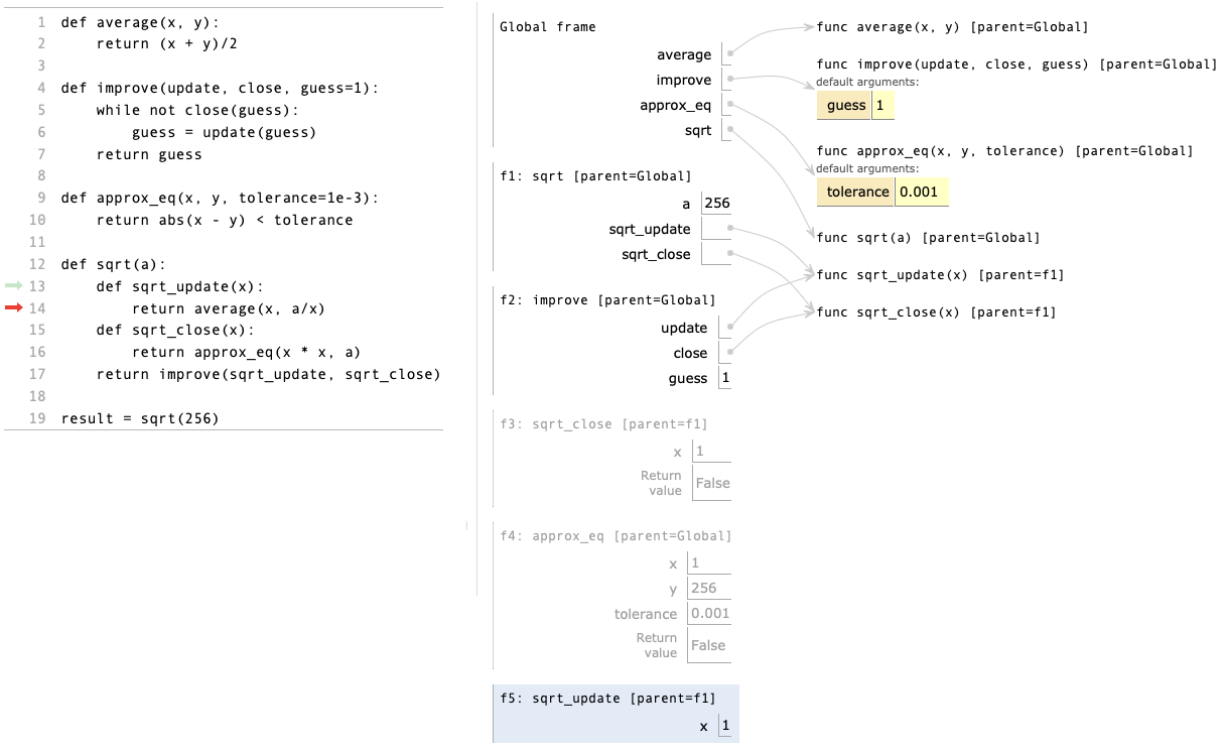
the environment first adds a local frame for `sqrt` and evaluates the `def` statements for `sqrt_update` and `sqrt_close`.

```
1  def average(x, y):
2      return (x + y)/2
3
4  def improve(update, close, guess=1):
5      while not close(guess):
6          guess = update(guess)
7      return guess
8
9  def approx_eq(x, y, tolerance=1e-3):
10     return abs(x - y) < tolerance
11
12 def sqrt(a):
13     def sqrt_update(x):
14         return average(x, a/x)
15     def sqrt_close(x):
16         return approx_eq(x * x, a)
17     return improve(sqrt_update, sqrt_close)
18
19 result = sqrt(256)
```



Function values each have a new annotation that we will include in environment diagrams from now on, a parent. The parent of a function value is the first frame of the environment in which that function was defined. Functions without parent annotations were defined in the global environment. When a user-defined function is called, the frame created has the same parent as that function.

Subsequently, the name `sqrt_update` resolves to this newly defined function, which is passed as an argument to `improve`. Within the body of `improve`, we must apply our `update` function (bound to `sqrt_update`) to the initial guess `x` of `1`. This final application creates an environment for `sqrt_update` that begins with a local frame containing only `x`, but with the parent frame `sqrt` still containing a binding for `a`.



The most critical part of this evaluation procedure is the transfer of the parent for `sqrt_update` to the frame created by calling `sqrt_update`. This frame is also annotated with `[parent=f1]`.

Extended Environments. An environment can consist of an arbitrarily long chain of frames, which always concludes with the global frame. Previous to this `sqrt` example, environments had at most two frames: a local frame and the global frame. By calling functions that were defined within other functions, via nested `def` statements, we can create longer chains. The environment for this call to `sqrt_update` consists of three frames: the local `sqrt_update` frame, the `sqrt` frame in which `sqrt_update` was defined (labeled `f1`), and the global frame.

The return expression in the body of `sqrt_update` can resolve a value for `a` by following this chain of frames. Looking up a name finds the first value bound to that name in the current environment. Python checks first in the `sqrt_update` frame — no `a` exists. Python checks next in the parent frame, `f1`, and finds a binding for `a` to `256`.

Hence, we realise two key advantages of lexical scoping in Python.

- The names of a local function do not interfere with names external to the function in which it is defined, because the local function name will be bound in the current local environment in which it was defined, rather than the global environment.
- A local function can access the environment of the enclosing function, because the body of the local function is evaluated in an environment that extends the evaluation environment in which it was defined.

The `sqrt_update` function carries with it some data: the value for `a` referenced in the environment in which it was defined. Because they "enclose" information in this way, locally defined functions are often called *closures*.

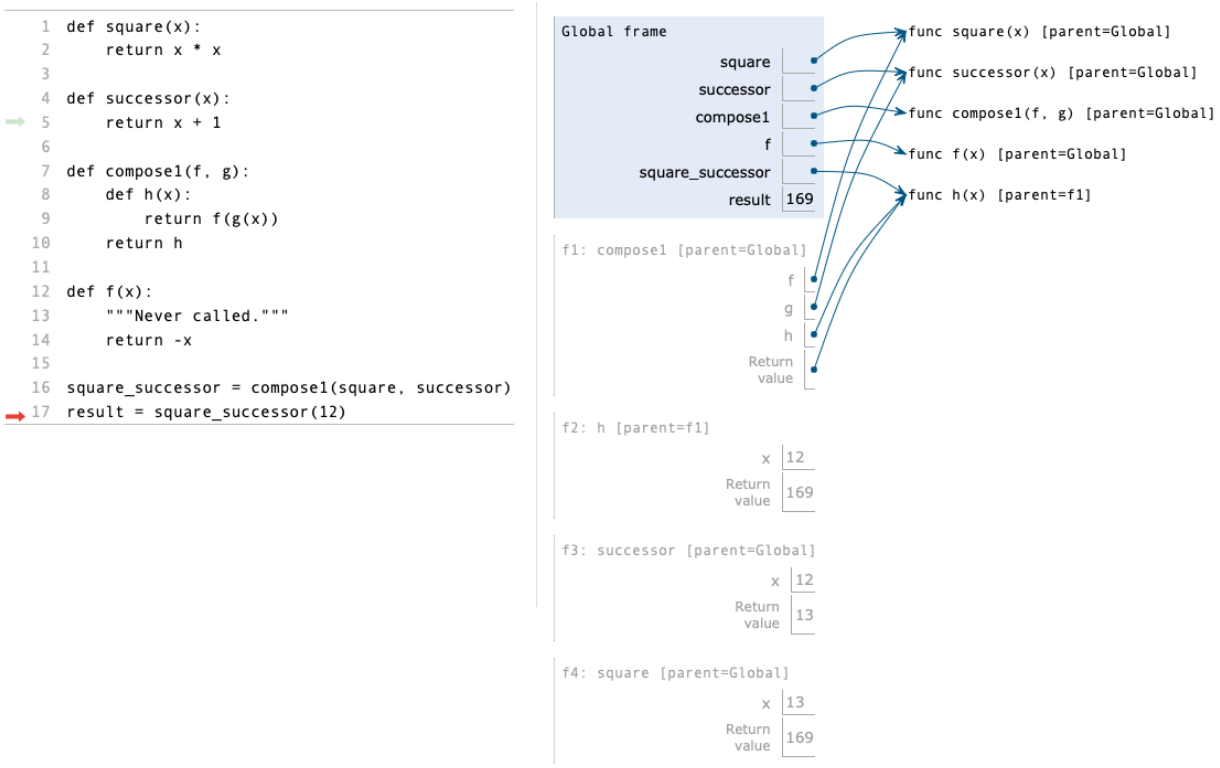
5.4 Functions as Returned Values

We can achieve even more expressive power in our programs by creating functions whose returned values are themselves functions. An important feature of lexically scoped programming languages is that locally defined functions maintain their parent environment when they are returned. The following example illustrates the utility of this feature.

Once many simple functions are defined, function composition is a natural method of combination to include in our programming language. That is, given two functions f(x) and g(x), we might want to define h(x) = f(g(x)). We can define function composition using our existing tools:

```
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h
```

The environment diagram for this example shows how the names f and g are resolved correctly, even in the presence of conflicting names.



The `1` in `compose1` is meant to signify that the composed functions all take a single argument. This naming convention is not enforced by the interpreter; the 1 is just part of the function name.

At this point, we begin to observe the benefits of our effort to define precisely the environment model of computation. No modification to our environment model is required to explain our ability to return functions in this way.

5.5 Example: Newton's Method

This extended example shows how function return values and local definitions can work together to express general ideas concisely. We will implement an algorithm that is used broadly in machine learning, scientific computing, hardware design, and optimisation.

Newton's method is a classic iterative approach to finding the arguments of a mathematical function that yield a return value of 0. These values are called the zeros of the function. Finding a zero of a function is often equivalent to solving some other problem of interest, such as computing a square root.

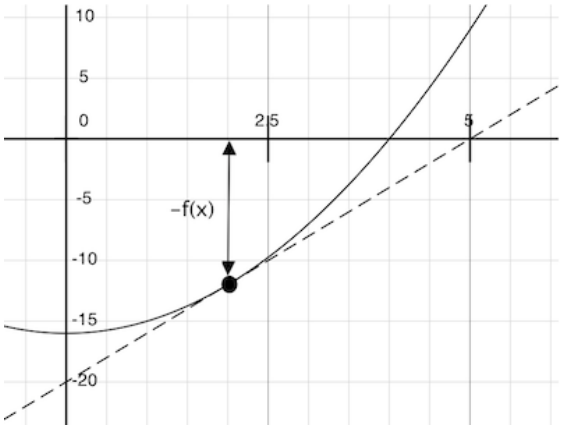
A motivating comment before we proceed: it is easy to take for granted the fact that we know how to compute square roots. Not just Python, but your phone, web browser, or pocket calculator can do so for you. However, part of learning computer science is

understanding how quantities like these can be computed, and the general approach presented here is applicable to solving a large class of equations beyond those built into Python.

Newton's method is an iterative improvement algorithm: it improves a guess of the zero for any function that is differentiable, which means that it can be approximated by a straight line at any point. Newton's method follows these linear approximations to find function zeros.

Imagine a line through the point $(x, f(x))$ that has the same slope as the curve for function $f(x)$ at that point. Such a line is called the *tangent*, and its slope is called the *derivative* of f at x.

This line's slope is the ratio of the change in function value to the change in function argument. Hence, translating x by $f(x)$ divided by the slope will give the argument value at which this tangent line touches 0.



A `newton_update` expresses the computational process of following this tangent line to 0, for a function `f` and its derivative `df`.

```
def newton_update(f, df):
    def update(x):
        return x - f(x) / df(x)
    return update
```

Finally, we can define the `find_root` function in terms of `newton_update`, our `improve` function, and a comparison to see if `f(x)` is near `0`.

```
def find_zero(f, df):
    def near_zero(x):
        return approx_eq(f(x), 0)
    return improve(newton_update(f, df), near_zero)
```

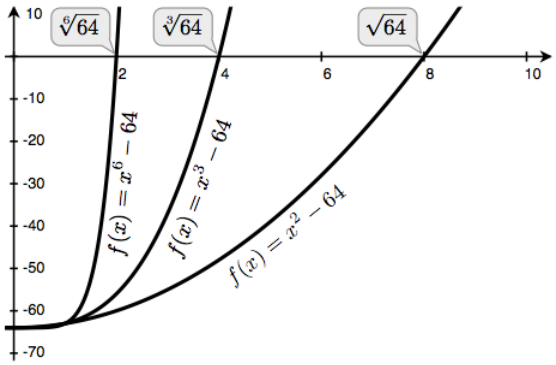
Computing Roots. Using Newton's method, we can compute roots of arbitrary degree n. The degree n root of a is x such that $x \cdot x \cdot x \ldots x = a$ with x repeated n times. For example,

- The square (second) root of 64 is 8, because $8 \cdot 8 = 64$.
- The cube (third) root of 64 is 4, because $4 \cdot 4 \cdot 4 = 64$.
- The sixth root of 64 is 2, because $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 64$.

We can compute roots using Newton's method with the following observations:

- The square root of 64 (written $\sqrt{64}$) is the value x such that $x^2 - 64 = 0$.
- More generally, the degree n root of a (written $\sqrt[n]{a}$) is the value x such that $x^n - a = 0$.

If we can find a zero of this last equation, then we can compute degree n roots. By plotting the curves for n equal to 2, 3, and 6 and a equal to 64, we can visualise this relationship.



We first implement `square_root` by defining `f` and its derivative `df`. We use from calculus the fact that the derivative of $f(x) = x^2 - a$ is the linear function $df(x) = 2x$.

```
def square_root_newton(a):
    def f(x):
        return x * x - a
    def df(x):
        return 2 * x
    return find_zero(f, df)

assert square_root_newton(64) == 8.0
```

Generalising to roots of arbitrary degree n, we compute $f(x) = x^n - a$ and its derivative $df(x) = n \cdot x^n - 1$. Note that `pow(x, n)` computes x^n.

```
def nth_root_of_a(n, a):
    def f(x):
        return pow(x, n) - a
    def df(x):
        return n * pow(x, n-1)
    return find_zero(f, df)

assert nth_root_of_a(2, 64) == 8.0
assert nth_root_of_a(3, 64) == 4.0
assert nth_root_of_a(6, 64) == 2.0
```

The approximation error in all of these computations can be reduced by changing the tolerance in `approx_eq` to a smaller number.

As you experiment with Newton's method, be aware that it will not always converge. The initial guess of `improve` must be sufficiently close to the zero, and various conditions about the function must be met. Despite this shortcoming, Newton's method is a powerful general computational method for solving differentiable equations. Very fast algorithms for logarithms and large integer division employ variants of the technique in modern computers.

5.6 Currying

We can use higher-order functions to convert a function that takes multiple arguments into a chain of functions that each take a single argument. More specifically, given a function `f(x, y)`, we can define a function `g` such that `g(x)(y)` is equivalent to `f(x, y)`. Here, `g` is a higher-order function that takes in a single argument `x` and returns another function that takes in a single argument `y`. This transformation is called *currying*.

As an example, we can define a curried version of the `pow` function:

```
def curried_pow(x):
    def h(y):
        return pow(x, y)
    return h

assert curried_pow(2)(3) == 8
```

Some programming languages, such as Haskell, only allow functions that take a single argument, so the programmer must curry all multi-argument procedures. In more general

languages such as Python, currying is useful when we require a function that takes in only a single argument. For example, the `map` pattern applies a single-argument function to a sequence of values. In later chapters, we will see more general examples of the `map` pattern, but for now, we can implement the pattern in a function:

```
def map_to_range(start, end, f):
    while start < end:
        print(f(start))
        start = start + 1
```

We can use `map_to_range` and `curried_pow` to compute the first ten powers of two, rather than specifically writing a function to do so:

```
>>> map_to_range(0, 10, curried_pow(2))
1
2
4
8
16
32
64
128
256
512
```

We can similarly use the same two functions to compute powers of other numbers. Currying allows us to do so without writing a specific function for each number whose powers we wish to compute.

In the above examples, we manually performed the currying transformation on the `pow` function to obtain `curried_pow`. Instead, we can define functions to automate currying, as well as the inverse uncurrying transformation:

```
def curry2(f):
    """Return a curried version of the given two-argument function."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g

def uncurry2(g):
```

```
    """Return a two-argument version of the given curried function."""
    def f(x, y):
        return g(x)(y)
    return f

pow_curried = curry2(pow)
```

```
>>> pow_curried(2)(5)
32
>>> map_to_range(0, 10, pow_curried(2))
1
2
4
8
16
32
64
128
256
512
```

The `curry2` function takes in a two-argument function `f` and returns a single-argument function `g`. When `g` is applied to an argument `x`, it returns a single-argument function `h`. When `h` is applied to `y`, it calls `f(x, y)`. Thus, `curry2(f)(x)(y)` is equivalent to `f(x, y)`. The `uncurry2` function reverses the currying transformation, so that `uncurry2(curry2(f))` is equivalent to `f`.

```
>>> uncurry2(pow_curried)(2, 5)
32
```

5.7 Lambda Expressions

So far, each time we have wanted to define a new function, we needed to give it a name. But for other types of expressions, we don't need to associate intermediate values with a name. That is, we can compute `a*b + c*d` without having to name the subexpressions `a*b` or `c*d`, or the full expression. In Python, we can create function values on the fly using lambda expressions, which evaluate to unnamed functions. A lambda expression evaluates to a function that has a single return expression as its body. Assignment and control statements are not allowed.

```
>>> def compose1(f, g):
        return lambda x: f(g(x))
```

We can understand the structure of a lambda expression by constructing a corresponding English sentence:

```
     lambda            x            :          f(g(x))
"A function that    takes x    and returns     f(g(x))"
```

The result of a lambda expression is called a lambda function. It has no intrinsic name (and so Python prints <lambda> for the name), but otherwise it behaves like any other function.

```
>>> s = lambda x: x * x
>>> s
<function <lambda> at 0xf3f490>
>>> s(12)
144
```

In an environment diagram, the result of a lambda expression is a function as well, named with the greek letter λ (lambda). Our compose example can be expressed quite compactly with lambda expressions.

```
def compose1(f, g):
    return lambda x: f(g(x))

f = compose1(lambda x: x * x,
             lambda y: y + 1)
result = f(12)
```

Some programmers find that using unnamed functions from lambda expressions to be shorter and more direct. However, compound lambda expressions are notoriously illegible, despite their brevity. The following definition is correct, but many programmers have trouble understanding it quickly.

```
compose1 = lambda f,g: lambda x: f(g(x))
```

In general, Python style prefers explicit `def` statements to lambda expressions, but allows them in cases where a simple function is needed as an argument or return value.

Such stylistic rules are merely guidelines; you can program any way you wish. However, as you write programs, think about the audience of people who might read your program one day. When you can make your program easier to understand, you do those people a favour.

The term lambda is a historical accident resulting from the incompatibility of written mathematical notation and the constraints of early type-setting systems.

> It may seem perverse to use lambda to introduce a procedure/function. The notation goes back to Alonzo Church, who in the 1930's started with a "hat" symbol; he wrote the square function as "$\hat{y}.y \times y$". But frustrated typographers moved the hat to the left of the parameter and changed it to a capital lambda: "$\Lambda y.y \times y$"; from there the capital lambda was changed to lowercase, and now we see "$\lambda y.y \times y$" in math books and `(lambda (y) (* y y))` in Lisp.
> — Peter Norvig (norvig.com/lispy2.html)

Despite their unusual etymology, lambda expressions and the corresponding formal language for function application, the lambda calculus, are fundamental computer science concepts shared far beyond the Python programming community.

5.8 Abstractions and First-Class Functions

We began this section with the observation that user-defined functions are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order functions permit us to manipulate these general methods to create further abstractions.

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs, build upon them, and generalise them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order functions is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the "rights and privileges" of first-class elements are:

- They may be bound to names.
- They may be passed as arguments to functions.
- They may be returned as the results of functions.
- They may be included in data structures.

Python awards functions full first-class status, and the resulting gain in expressive power is enormous.

5.9 Function Decorators

Python provides special syntax to apply higher-order functions as part of executing a def statement, called a decorator. Perhaps the most common example is a trace.

```
def trace(fn):
    def wrapped(x):
        print('-> ', fn, '(', x, ')')
        return fn(x)
    return wrapped

@trace
def triple(x):
    return 3 * x
```

```
>>> triple(12)
->  <function triple at 0x102a39848> ( 12 )
36
```

In this example, a higher-order function `trace` is defined, which returns a function that precedes a call to its argument with a print statement that outputs the argument. The `def` statement for triple has an annotation, `@trace`, which affects the execution rule for `def`. As usual, the function `triple` is created. However, the name `triple` is not bound to this function. Instead, the name `triple` is bound to the returned function value of calling `trace` on the newly defined triple function. In code, this decorator is equivalent to:

```
def triple(x):
    return 3 * x
triple = trace(triple)
```

Extra for experts. The decorator symbol `@` may also be followed by a call expression. The expression following `@` is evaluated first (just as the name `trace` was evaluated above), the `def` statement second, and finally the result of evaluating the decorator expression is applied to the newly defined function, and the result is bound to the name in the `def` statement.

6 Recursive Functions

A function is called *recursive* if the body of the function calls the function itself, either directly or indirectly. That is, the process of executing the body of a recursive function

may in turn require applying that function again. Recursive functions do not use any special syntax in Python, but they do require some effort to understand and create.

We'll begin with an example problem: write a function that sums the digits of a natural number. When designing recursive functions, we look for ways in which a problem can be broken down into simpler problems. In this case, the operators `%` and `//` can be used to separate a number into two parts: its last digit and all but its last digit.

```
>>> 18117 % 10
7
>>> 18117 // 10
1811
```

The sum of the digits of 18117 is 1+8+1+1+7 = 18. Just as we can separate the number, we can separate this sum into the last digit, 7, and the sum of all but the last digit, 1+8+1+1 = 11. This separation gives us an algorithm: to sum the digits of a number n, add its last digit `n % 10` to the sum of the digits of `n // 10`. There's one special case: if a number has only one digit, then the sum of its digits is itself. This algorithm can be implemented as a recursive function.

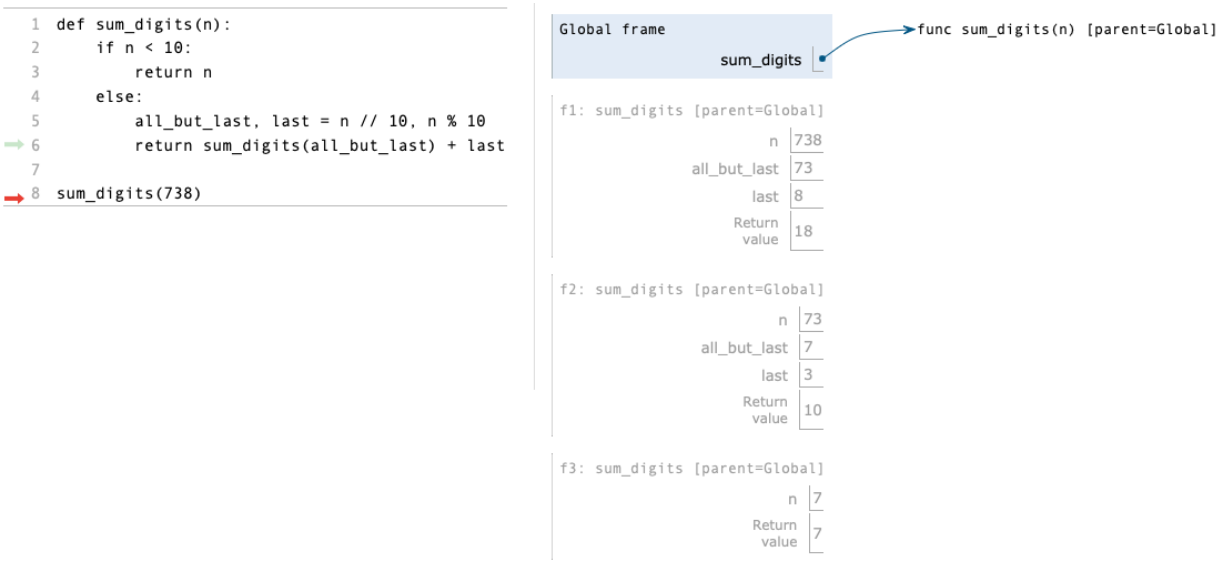
```
def sum_digits(n):
    """Return the sum of the digits of positive int n.

    result is a positive int
    """
    if n < 10:
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```

This definition of `sum_digits` is both complete and correct, even though the `sum_digits` function is called within its own body. The problem of summing the digits of a number is broken down into two steps: summing all but the last digit, then adding the last digit. Both of these steps are simpler than the original problem. The function is recursive because the first step is the same kind of problem as the original problem. That is, `sum_digits` is exactly the function we need in order to implement `sum_digits`.

```
>>> sum_digits(9)
9
>>> sum_digits(18117)
```

```
18
>>> sum_digits(9437184)
36
>>> sum_digits(114088554020540646134703288483384)
126
```

We can understand precisely how this recursive function works using our environment model of computation. No new rules are required.



When the `def` statement is executed, the name `sum_digits` is bound to a new function, but the body of that function is not yet executed. Therefore, the circular nature of `sum_digits` is not a problem yet. Then, `sum_digits` is called on `738`:

1. A local frame for `sum_digits` with `n` bound to `738` is created, and the body of `sum_digits` is executed in the environment that starts with that frame.
2. Since `738` is not less than `10`, the assignment statement on line 5 is executed, splitting `738` into `73` and `8`.
3. In the following `return` statement, `sum_digits` is called on `73`, the value of `all_but_last` in the current environment.
4. Another local frame for `sum_digits` is created, this time with `n` bound to `73`. The body of `sum_digits` is again executed in the new environment that starts with this frame.
5. Since `73` is also not less than `10`, `73` is split into `7` and `3` and `sum_digits` is called on `7`, the value of `all_but_last` evaluated in this frame.

6. A third local frame for `sum_digits` is created, with `n` bound to `7`.
7. In the environment starting with this frame, it is true that `n < 10`, and therefore `7` is returned.
8. In the second local frame, this return value `7` is summed with `3`, the value of `last`, to return `10`.
9. In the first local frame, this return value `10` is summed with `8`, the value of `last`, to return `18`.

This recursive function works correctly, despite its circular character, because it is applied three times, but with a different argument each time. Moreover, the second application was a simpler instance of the digit summing problem than the first. Generate the environment diagram for the call `sum_digits(18117)` to see that each successive call to `sum_digits` takes a smaller argument than the last, until eventually a single-digit input is reached.

This example also illustrates how functions with simple bodies can evolve complex computational processes by using recursion.

6.1 Recursive Function Definitions

A common pattern can be found in the body of many recursive functions. The body begins with a base case, a conditional statement that defines the behaviour of the function for the inputs that are simplest to process. In the case of `sum_digits`, the base case is any single-digit argument, and we simply return that argument. Some recursive functions will have multiple base cases.

The base cases are then followed by one or more recursive calls. Recursive calls always have a certain character: they simplify the original problem. Recursive functions express computation by simplifying problems incrementally. For example, summing the digits of 7 is simpler than summing the digits of 73, which in turn is simpler than summing the digits of 738. For each subsequent call, there is less work left to be done.

Recursive functions often solve problems in a different way than the iterative approaches that we have used previously. Consider a function `fact` to compute n factorial, where for example `fact(4)` computes $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

A natural implementation using a `while` statement accumulates the `total` by multiplying together each positive integer up to n.

```
def fact_iter(n):
    total = 1
    k = 1
```

```
    while k <= n:
        total = total * k
        k = k + 1
    return total

assert fact_iter(4) == 24
```

On the other hand, a recursive implementation of factorial can express `fact(n)` in terms of `fact(n-1)`, a simpler problem. The base case of the recursion is the simplest form of the problem: `fact(1)` is `1`.

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)

assert fact(4) == 24
```

These two factorial functions differ conceptually. The iterative function constructs the result from the base case of 1 to the final total by successively multiplying in each term. The recursive function, on the other hand, constructs the result directly from the final term, `n`, and the result of the simpler problem, `fact(n-1)`.

As the recursion "unwinds" through successive applications of the `fact` function to simpler and simpler problem instances, the result is eventually built starting from the base case. The recursion ends by passing the argument 1 to `fact`; the result of each call depends on the next until the base case is reached.

The correctness of this recursive function is easy to verify from the standard definition of the mathematical function for factorial:

$$\begin{aligned}(n-1)! &= (n-1)\cdot(n-2)\cdot\cdots\cdot 1\\ n! &= n\cdot(n-1)\cdot(n-2)\cdot\cdots\cdot 1\\ n! &= n\cdot(n-1)!\end{aligned}$$

While we can unwind the recursion using our model of computation, it is often clearer to think about recursive calls as functional abstractions. That is, we should not care about how `fact(n-1)` is implemented in the body of `fact`; we should simply trust that it computes the factorial of `n-1`. Treating a recursive call as a functional abstraction has been called a recursive leap of faith. We define a function in terms of itself, but

simply trust that the simpler cases will work correctly when verifying the correctness of the function. In this example, we trust that `fact(n-1)` will correctly compute $(n-1)!$; we must only check that $n!$ is computed correctly if this assumption holds. In this way, verifying the correctness of a recursive function is a form of proof by induction.

The functions `fact_iter` and `fact` also differ because the former must introduce two additional names, `total` and `k`, that are not required in the recursive implementation. In general, iterative functions must maintain some local state that changes throughout the course of computation. At any point in the iteration, that state characterises the result of completed work and the amount of work remaining. For example, when `k` is 3 and `total` is 2, there are still two terms remaining to be processed, 3 and 4. On the other hand, `fact` is characterised by its single argument `n`. The state of the computation is entirely contained within the structure of the environment, which has return values that take the role of `total`, and binds `n` to different values in different frames rather than explicitly tracking `k`.

Recursive functions leverage the rules of evaluating call expressions to bind names to values, often avoiding the nuisance of correctly assigning local names during iteration. For this reason, recursive functions can be easier to define correctly. However, learning to recognise the computational processes evolved by recursive functions certainly requires practice.

6.2 Mutual Recursion

When a recursive procedure is divided among two functions that call each other, the functions are said to be *mutually* recursive. As an example, consider the following definition of `even` and `odd` for non-negative integers:

- a number is even if it is one more than an odd number
- a number is odd if it is one more than an even number
- 0 is even

Using this definition, we can implement mutually recursive functions to determine whether a number is even or odd:

```
def is_even(n):
    """Determine wether given number is even.

    n is int >=0
    result is bool
    """
```

```
    if n == 0:
        return True
    else:
        return is_odd(n-1)

assert is_even(0)
assert is_even(8)

def is_odd(n):
    """Determine wether given number is odd.

    n is int >=0
    result is bool
    """
    if n == 0:
        return False
     else:
        return is_even(n-1)

assert is_odd(1)
assert is_odd(15)
```

As such, mutual recursion is no more mysterious or powerful than simple recursion, and it provides a mechanism for maintaining abstraction within a complicated recursive program.

Remark: Our definitions are a simple example for mutual recursion. Using `n % 2 == 0` and `n % 2 == 1` instead would be more efficient.

6.3 Printing in Recursive Functions

The computational process evolved by a recursive function can often be visualised using calls to `print`. As an example, we will implement a function `cascade` that prints all prefixes of a number from largest to smallest to largest.

```
def cascade(n):
    """Print a cascade of prefixes of n.

    n is int >= 0
    """
    if n < 10:
```

```
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```
>>> cascade(2013)
2013
201
20
2
20
201
2013
```

In this recursive function, the base case is a single-digit number, which is printed. Otherwise, a recursive call is placed between two calls to print.

It is not a rigid requirement that base cases be expressed before recursive calls. In fact, this function can be expressed more compactly by observing that `print(n)` is repeated in both clauses of the conditional statement, and therefore can precede it.

```
def cascade(n):
    """Print a cascade of prefixes of n.

    n is int >= 0
    """
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

As another example of mutual recursion, consider a two-player game in which there are `n` initial pebbles on a table. The players take turns, removing either one or two pebbles from the table, and the player who removes the final pebble wins. Suppose that Alice and Bob play this game, each using a simple strategy:

- Alice always removes a single pebble.
- Bob removes two pebbles if an even number of pebbles is on the table, and one otherwise.

Given `n` initial pebbles and Alice starting, who wins the game?

A natural decomposition of this problem is to encapsulate each strategy in its own function. This allows us to modify one strategy without affecting the other, maintaining the abstraction barrier between the two. In order to incorporate the turn-by-turn nature of the game, these two functions call each other at the end of each turn.

```
def play_alice(n):
    if n == 0:
        print("Bob wins!")
    else:
        play_bob(n-1)
def play_bob(n):
    if n == 0:
        print("Alice wins!")
    elif is_even(n):
        play_alice(n-2)
    else:
        play_alice(n-1)
```

```
>>> play_alice(20)
Bob wins!
```

In `play_bob`, we see that multiple recursive calls may appear in the body of a function. However, in this example, each call to `play_bob` calls `play_alice` at most once. In the next section, we consider what happens when a single function call makes multiple direct recursive calls.

6.4 Tree Recursion

Another common pattern of computation is called tree recursion, in which a function calls itself more than once. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two.

```
1 def fib(n):
2     if n == 1:
3         return 0
4     if n == 2:
5         return 1
6     else:
7         return fib(n-2) + fib(n-1)
```

```
8
9 result = fib(6)
```

This recursive definition is tremendously appealing relative to our previous attempts: it exactly mirrors the familiar definition of Fibonacci numbers. A function with multiple recursive calls is said to be tree recursive because each call branches into multiple smaller calls, each of which branches into yet smaller calls, just as the branches of a tree become smaller but more numerous as they extend from the trunk.

We were already able to define a function to compute Fibonacci numbers without tree recursion. In fact, our previous attempts were more efficient. Next, we consider a problem for which the tree recursive solution is substantially simpler than any iterative alternative.

6.5 Example: Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order. For example, the number of partitions of 6 using parts up to 4 is 9.

1. $6 = 2 + 4$
2. $6 = 1 + 1 + 4$
3. $6 = 3 + 3$
4. $6 = 1 + 2 + 3$
5. $6 = 1 + 1 + 1 + 3$
6. $6 = 2 + 2 + 2$
7. $6 = 1 + 1 + 2 + 2$
8. $6 = 1 + 1 + 1 + 1 + 2$
9. $6 = 1 + 1 + 1 + 1 + 1 + 1$

We will define a function `count_partitions(n, m)` that returns the number of different partitions of `n` using parts up to `m`. This function has a simple solution as a tree-recursive function, based on the following observation:

The number of ways to partition `n` using integers up to `m` equals

1. the number of ways to partition n-m using integers up to m, and
2. the number of ways to partition n using integers up to m-1.

To see why this is true, observe that all the ways of partitioning n can be divided into two groups: those that include at least one m and those that do not. Moreover, each partition in the first group is a partition of n-m, followed by m added at the end. In the example above, the first two partitions contain 4, and the rest do not.

Therefore, we can recursively reduce the problem of partitioning n using integers up to m into two simpler problems: (1) partition a smaller number n-m, and (2) partition with smaller components up to m-1.

To complete the implementation, we need to specify the following base cases:

1. There is one way to partition 0: include no parts.
2. There are 0 ways to partition a negative n.
3. There are 0 ways to partition any n greater than 0 using parts of size 0 or less.

```
def count_partitions(n, m):
    """Count the ways to partition n using parts up to m.

    n is int
    m is int >= 0
    """
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        return count_partitions(n-m, m) + count_partitions(n, m-1)

assert count_partitions(6, 4) == 9
assert count_partitions(5, 5) == 7
assert count_partitions(10, 10) == 42
assert count_partitions(15, 15) == 176
assert count_partitions(20, 20) == 627
```

We can think of a tree-recursive function as exploring different possibilities. In this case, we explore the possibility that we use a part of size m and the possibility that we do not. The first and second recursive calls correspond to these possibilities.

Implementing this function without recursion would be substantially more involved. Interested readers are encouraged to try.

6.6 Recursion vs. Iteration

Every iterative function definition can be converted into a recursive function definition. Most recursive function definitions can be converted into iterative function definitions (such as `fact` or `sum_digits`; but in the next subsections we give examples where that is difficult). As we usually have the choice, which form of definition should we use? In imperative programming languages with a focus on state and statements, iteration is generally the better and more efficient solution unless recursion is required. In functional programming languages recursion is usually more efficient and some such languages such as Haskell do not even support iteration. While Python aims for the middle ground, iterative solutions — where possible — are usually better than recursive ones.

While functional programming languages allow arbitrarily deep recursion (as long as the memory for environment frames suffices), the Python interpreter has a limit for maximum recursion depth:

```
>>> def f(x):
    return f(x+1)

>>> f(0)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    f(0)
  File "<pyshell#3>", line 2, in f
    return f(x+1)
  File "<pyshell#3>", line 2, in f
    return f(x+1)
  File "<pyshell#3>", line 2, in f
    return f(x+1)
  [Previous line repeated 1023 more times]
RecursionError: maximum recursion depth exceeded
```

So when a desired iteration can easily be achieved with a loop statement, then Python programmers prefer a loop to recursion. Recursion is mainly used for tree recursion (where recursion depth will also generally be lower).

Index