Chapter 1: Building Abstractions with Functions

Contents

1.1 Introduction

Computer science is a tremendously broad academic discipline. The areas of globally distributed systems, artificial intelligence, robotics, graphics, security, scientific computing, computer architecture, and dozens of emerging subfields each 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 tremendous productivity of computer science is only possible because it 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.

These fundamental ideas have long been taught at Berkeley using the classic textbook *Structure and Interpretation of Computer Programs* (SICP) by Harold Abelson and Gerald Jay Sussman with Julie Sussman. These lecture notes borrow heavily from that textbook, which the original authors have kindly licensed for adaptation and reuse.

The embarkment of our intellectual journey requires no revision, nor should we expect that it ever will.

> We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.
>
> The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.
>
> A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring.
>
> —Abelson and Sussman, SICP (1993)

1.1.1 Programming in Python

> A language isn't something you learn so much as something you join.
>
> —Arika Okrent

In order to define computational processes, we need a programming language; preferably one many humans and a great variety of computers can all understand. In this course, we will learn the Python language.

Python is a widely used programming language that has recruited enthusiasts from many professions: 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 code and experiences, and collectively develop software and tools. Dedicated members often achieve celebrity and widespread esteem for their contributions. Perhaps someday you will be named among these elite Pythonistas.

The Python language itself is the product of a large volunteer community that prides itself on the diversity of its contributors. The language was conceived and first implemented by Guido van Rossum in the late 1980's. The first chapter of his Python 3 Tutorial explains why Python is so popular, among the many languages available today.

Python excels as an instructional language because, throughout its history, Python's developers have emphasized the human interpretability of Python code, reinforced by the Zen of Python guiding principles of beauty, simplicity, and readability. Python is particularly appropriate for this course because its broad set of features support a variety of different programming styles, which we will explore. While there is no single way to program in Python, there are a set of conventions shared across the developer community that facilitate the process of reading, understanding, and extending existing programs. Hence, Python's combination of great flexibility and accessibility allows students to explore many programming paradigms, and then apply their newly acquired knowledge to thousands of ongoing projects.

These notes maintain the spirit of SICP by introducing the features of Python in lock step with techniques for abstraction design and a rigorous model of computation. In addition, these notes provide a practical introduction to Python programming, including some advanced language features and illustrative examples. Learning Python will come naturally as you progress through the course.

However, Python is a rich language with many features and uses, and we consciously introduce them slowly as we layer on fundamental computer science concepts. For experienced students who want to inhale all of the details of the language quickly, we recommend reading Mark Pilgrim's book Dive Into Python 3, which is freely available online. The topics in that book differ substantially from the topics of this course, but the book contains very valuable practical information on using the Python language. Be forewarned: unlike these notes, Dive Into Python 3 assumes substantial programming experience.

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.

1.1.2 Installing Python 3

As with all great software, Python has many versions. This course will use the most recent stable version of Python 3 (currently Python 3.2). Many computers have older versions of Python installed already, but those will not suffice for this course. You should be able to use any computer for this course, but expect to install Python 3. Don't worry, Python is free.

Dive Into Python 3 has detailed installation instructions for all major platforms. These instructions mention Python 3.1 several times, but you're better off with Python 3.2 (although the differences are insignificant for this course). All instructional machines in the EECS department have Python 3.2 already installed.

1.1.3 Interactive Sessions

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

There are several ways to start an interactive session, and they differ in their properties. Try them all to find out what you prefer. They all use exactly the same interpreter behind the scenes.

- The simplest and most common way is to run the Python 3 application. Type `python3` at a terminal prompt (Mac/Unix/Linux) or open the Python 3 application in Windows.
- A more user-friendly application for those learning the language is called Idle 3 (`idle3`). Idle colorizes your code (called syntax highlighting), pops up usage hints, and marks the source of some errors. Idle is always bundled with Python, so you have already installed it.
- The Emacs editor can run an interactive session inside one of its buffers. While slightly more challenging to learn, Emacs is a powerful and versatile editor for any programming language. Read the 61A Emacs Tutorial to get started. Many programmers who invest the time to learn Emacs never switch editors again.

In any case, 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
```

Controls: Each session keeps a history of what you have typed. To access that history, press `<Control>-P` (previous) and `<Control>-N` (next). `<Control>-D` exits a session, which discards this history.

1.1.4 First Example

And, as imagination bodies forth
The forms of things to unknown, and the poet's pen
Turns them to shapes, and gives to airy nothing

A local habitation and a name.

—William Shakespeare, A Midsummer-Night’s Dream

To give Python the introduction it deserves, we will begin with an example that uses several language features. In the next section, we will have to start from scratch and build up the language piece by piece. Think of this section as a sneak preview of powerful features to come.

Python has built-in support for a wide range of common programming activities, like manipulating text, displaying graphics, and communicating over the Internet. The import statement

```
>>> from urllib.request import urlopen
```

loads functionality for accessing data on the Internet. In particular, it makes available a function called `urlopen`, which can access the content at a uniform resource locator (URL), which is a location of something on the Internet.

Statements & Expressions. Python code consists of statements and expressions. Broadly, computer programs consist of instructions to either

1. Compute some value

2. Carry out some action

Statements typically describe actions. When the Python interpreter executes a statement, it carries out the corresponding action. On the other hand, expressions typically describe computations that yield values. When Python evaluates an expression, it computes its value. This chapter introduces several types of statements and expressions.

The assignment statement

```
>>> shakespeare = urlopen(’http://inst.eecs.berkeley.edu/~cs61a/fa11/shakespeare.txt
```

associates the name `shakespeare` with the value of the expression that follows. That expression applies the `urlopen` function to a URL that contains the complete text of William Shakespeare’s 37 plays, all in a single text document.

Functions. Functions encapsulate logic that manipulates data. A web address is a piece of data, and the text of Shakespeare’s plays is another. The process by which the former leads to the latter may be complex, but we can apply that process using only a simple expression because that complexity is tucked away within a function. Functions are the primary topic of this chapter.

Another assignment statement

```
>>> words = set(shakespeare.read().decode().split())
```

associates the name `words` to the set of all unique words that appear in Shakespeare’s plays, all 33,721 of them. The chain of commands to `read`, `decode`, and `split`, each operate on an intermediate computational entity: data is read from the opened URL, that data is decoded into text, and that text is split into words. All of those words are placed in a `set`.

Objects. A set is a type of object, one that supports set operations like computing intersections and testing membership. An object seamlessly bundles together data and the logic that manipulates that data, in a way that hides the complexity of both. Objects are the primary topic of Chapter 2.

The expression

```
>>> {w for w in words if len(w) >= 5 and w[::-1] in words}
{’madam’, ’stink’, ’leets’, ’rever’, ’drawer’, ’stops’, ’sessa’,
’repaid’, ’speed’, ’redder’, ’devil’, ’minim’, ’spots’, ’asses’,
’refer’, ’lived’, ’keels’, ’diaper’, ’sleek’, ’steel’, ’leper’,
’level’, ’deeps’, ’repel’, ’reward’, ’knits’}
```

is a compound expression that evaluates to the set of Shakespearian words that appear both forward and in reverse. The cryptic notation `w[::-1]` enumerates each letter in a word, but the `-1` says to step backwards (`::` here means that the positions of the first and last characters to enumerate are defaulted.) When you enter an expression in an interactive session, Python prints its value on the following line, as shown.

Interpreters. Evaluating compound expressions requires a precise procedure that interprets code in a predictable way. A program that implements such a procedure, evaluating compound expressions and statements, is called an interpreter. The design and implementation of interpreters is the primary topic of Chapter 3.

When compared with other computer programs, interpreters for programming languages are unique in their generality. Python was not designed with Shakespeare or palindromes in mind. However, its great flexibility allowed us to process a large amount of text with only a few lines of code.

In the end, we will find that all of these core concepts are closely related: functions are objects, objects are functions, and interpreters are instances of both. However, developing a clear understanding of each of these concepts and their role in organizing code is critical to mastering the art of programming.

1.1.5 Practical Guidance: Errors

Python is waiting for your command. You are encouraged to experiment with the language, even though you may not yet know its full vocabulary and structure. However, be prepared for errors. While computers are tremendously fast and flexible, they are also extremely rigid. The nature of computers is described in Stanford's introductory course as

> The fundamental equation of computers is: `computer = powerful + stupid`
>
> Computers are very powerful, looking at volumes of data very quickly. Computers can perform billions of operations per second, where each operation is pretty simple.
>
> Computers are also shockingly stupid and fragile. The operations that they can do are extremely rigid, simple, and mechanical. The computer lacks anything like real insight .. it's nothing like the HAL 9000 from the movies. If nothing else, you should not be intimidated by the computer as if it's some sort of brain. It's very mechanical underneath it all.
>
> Programming is about a person using their real insight to build something useful, constructed out of these teeny, simple little operations that the computer can do.
>
> —Francisco Cai and Nick Parlante, Stanford CS101

The rigidity of computers will immediately become apparent as you experiment with the Python interpreter: even the smallest spelling and formatting changes will cause unexpected outputs and errors.

Learning to interpret errors and diagnose the cause of unexpected errors is called *debugging*. Some guiding principles of debugging are:

1. **Test incrementally**: Every well-written program is composed of small, modular components that can be tested individually. Test everything you write as soon as possible to catch errors early and gain confidence in your components.

2. **Isolate errors**: An error in the output of a compound program, expression, or statement can typically be attributed to a particular modular component. When trying to diagnose a problem, trace the error to the smallest fragment of code you can before trying to correct it.

3. **Check your assumptions**: Interpreters do carry out your instructions to the letter --- no more and no less. Their output is unexpected when the behavior of some code does not match what the programmer believes (or assumes) that behavior to be. Know your assumptions, then focus your debugging effort on verifying that your assumptions actually hold.

4. **Consult others**: You are not alone! If you don't understand an error message, ask a friend, instructor, or search engine. If you have isolated an error, but can't figure out how to correct it, ask someone else to take a look. A lot of valuable programming knowledge is shared in the context of team problem solving.

Incremental testing, modular design, precise assumptions, and teamwork are themes that persist throughout this course. Hopefully, they will also persist throughout your computer science career.

1.2 The Elements of Programming

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 organize our ideas about 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 mechanisms for accomplishing this:

- **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.

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 and should have methods for combining and abstracting both functions and data.

1.2.1 Expressions

Having experimented with the full Python interpreter, we now must start anew, methodically developing the Python language piece by piece. Be patient if the examples seem simplistic --- more exciting material is soon to come.

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:

```
>>> -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.

1.2.2 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. A function in Python is more than just an input-output mapping; it describes a computational process. However, the way in which Python expresses function application is the same as in mathematics.

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

This call expression has subexpressions: the operator precedes parentheses, which enclose a comma-delimited list of operands. The operator must be a function. The operands can be any values; in this case they are numbers.

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 several 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 others who may read your code in the future.

Finally, 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.

1.2.3 Importing Library Functions

Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make their names available by default, so as to avoid complete chaos. Instead, it organizes 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, exp
>>> sqrt(256)
16.0
>>> exp(1)
2.718281828459045
```

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 or exp).

The Python 3 Library Docs 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 behavior than reading the documemtation. As you become familiar with the Python language and vocabulary, this documentation will become a valuable reference source.

1.2.4 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
```

We can also assign multiple values to multiple names in a single statement, where names and expressions are separated by commas.

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

The = symbol is called the *assignment* operator in Python (and many other languages). Assignment is Python's 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 print 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(3, 4)
4
```

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

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

In Python, the names bound via assignment are often called *variable names* because they can be bound to a variety of different values in the course of executing a program.

1.2.5 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:

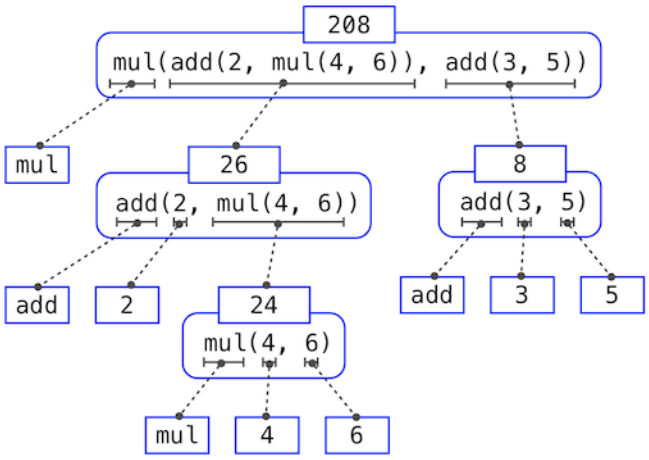
1. Evaluate the operator and operand subexpressions, then
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

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

```
>>> mul(add(2, mul(4, 6)), add(3, 5))
208
```

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



This illustration is called an *expression tree*. In computer science, trees grow from the top down. 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, 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.

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 statement or expression has its own evaluation or execution procedure, which we will introduce incrementally as we proceed.

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 loosely say that numerals (and expressions) themselves evaluate to values in the context of Python programs.

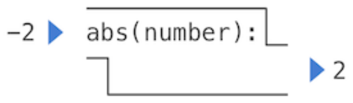
1.2.6 Function Diagrams

As we continue to develop a formal model of evaluation, we will find that diagramming the internal state of the interpreter helps us track the progress of our evaluation procedure. An essential part of these diagrams is a representation of a function.

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

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

can be depicted as a small machine that takes input and produces output.

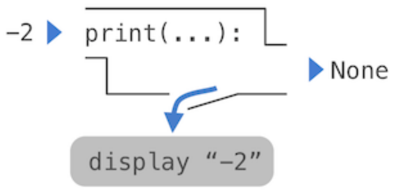


The function `abs` is *pure*. Pure functions have the property that applying them has no effects beyond returning a value.

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

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 names of its arguments. The function `abs` takes only one argument called `number`; providing more or fewer will result in an error. The function `print` can take an arbitrary number of arguments, hence its rendering as `print(...)`. A description of the arguments that a function can take is called the function's *signature*.

1.3 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 built-in data and functions.
2. Nested function application provides a means of combining operations.
3. Binding names to values provides a limited means of abstraction.

Now we will learn about *function definitions*, a much more powerful abstraction technique by which a name can be bound to 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`.

Function definitions consist of a `def` statement that indicates a `<name>` and a 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! Convention dictates that we indent with four spaces, rather than a tab. 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. (Soon, we will see that the indented region can span multiple lines.)

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:

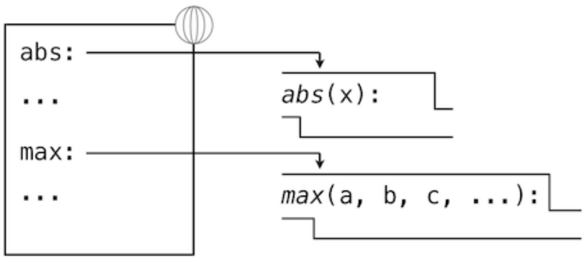
```
>>> def sum_squares(x, y):
        return add(square(x), square(y))

>>> 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.

1.3.1 Environments

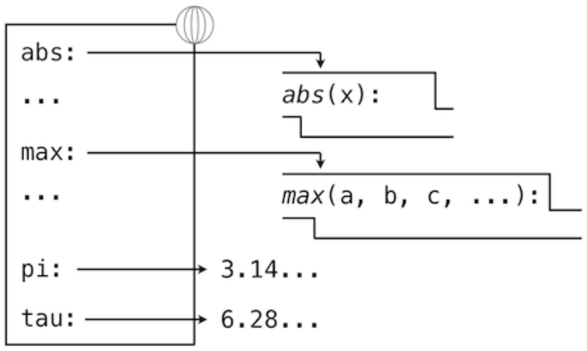
Our subset of Python is now complex enough that the meaning of programs is non-obvious. What if a formal parameter has the same name as a built-in function? Can two functions share names without confusion? To resolve such questions, we must describe environments in more detail.

An environment in which an expression is evaluated consists of a sequence of *frames*, depicted as boxes. Each frame contains *bindings*, which associate a name with its corresponding value. There is a single *global* frame that contains name bindings for all built-in functions (only `abs` and `max` are shown). We indicate the global frame with a globe symbol.

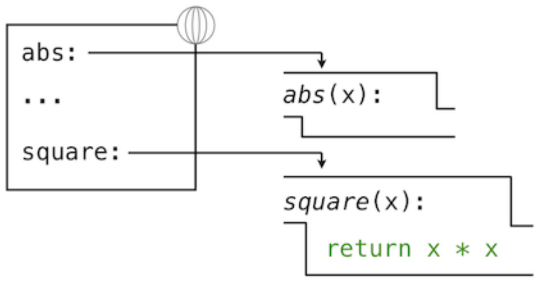


Assignment and import statements add entries to the first frame of the current environment. So far, our environment consists only of the global frame.

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



A `def` statement also binds a name to the function created by the definition. The resulting environment after defining `square` appears below:



These *environment diagrams* show the bindings of the current environment, along with the values (which are not part of any frame) to which names are bound. Notice that the name of a function is repeated, once in the frame, and once as part of the function itself. This repetition is intentional: many different names may refer to the same function, but that function itself has only one intrinsic name. However, looking up the value for a name in an environment only inspects name bindings. The intrinsic name of a function **does not** play a role in looking up names. In the example we saw earlier,

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

The name *max* is the intrinsic name of the function, and that's what you see printed as the value for `f`. In addition, both the names `max` and `f` are bound to that same function in the global environment.

As we proceed to introduce additional features of Python, we will have to extend these diagrams. Every time we do, we will list the new features that our diagrams can express.

New environment Features: Assignment and user-defined function definition.

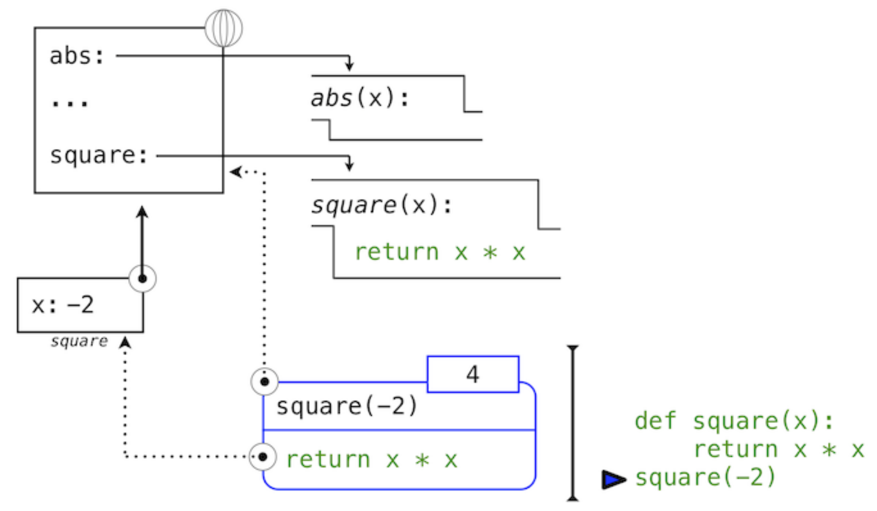
1.3.2 Calling User-Defined Functions

To evaluate a call expression whose operator names a user-defined function, the Python interpreter follows a process similar to the one for evaluating expressions with a built-in operator function. That is, the interpreter evaluates the operand expressions, and then applies the named function to the resulting arguments.

The act of 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:

1. Bind the arguments to the names of the function's formal parameters in a new *local* frame.
2. Evaluate the body of the function in the environment beginning at that frame and ending at the global frame.

The environment in which the body is evaluated consists of two frames: first the local frame that contains argument bindings, then the global frame that contains everything else. Each instance of a function application has its own independent local frame.



This figure includes two different aspects of the Python interpreter: the current environment, and a part of the expression tree related to the current line of code being evaluated. We have depicted the evaluation of a call expression that has a user-defined function (in blue) as a two-part rounded rectangle. Dotted arrows indicate which environment is used to evaluate the expression in each part.

- The top half shows the call expression being evaluated. This call expression is not internal to any function, so it is evaluated in the global environment. Thus, any names within it (such as `square`) are looked up in the global frame.
- The bottom half shows the body of the `square` function. Its return expression is evaluated in the new environment introduced by step 1 above, which binds the name of `square`'s formal parameter `x` to the value of its argument, `-2`.

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:

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

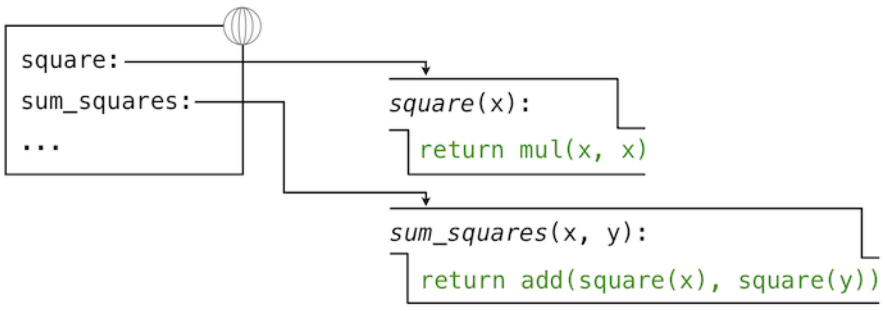
Our conceptual framework of environments, names, and functions constitutes a *model of evaluation*; while some mechanical details are still unspecified (e.g., how a binding is implemented), our model does precisely and correctly describe how the interpreter evaluates call expressions. In Chapter 3 we shall see how this model can serve as a blueprint for implementing a working interpreter for a programming language.

New environment Feature: Function application.

1.3.3 Example: Calling a User-Defined Function

Let us again consider our two simple definitions:

```
>>> from operator import add, mul
>>> def square(x):
        return mul(x, x)

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

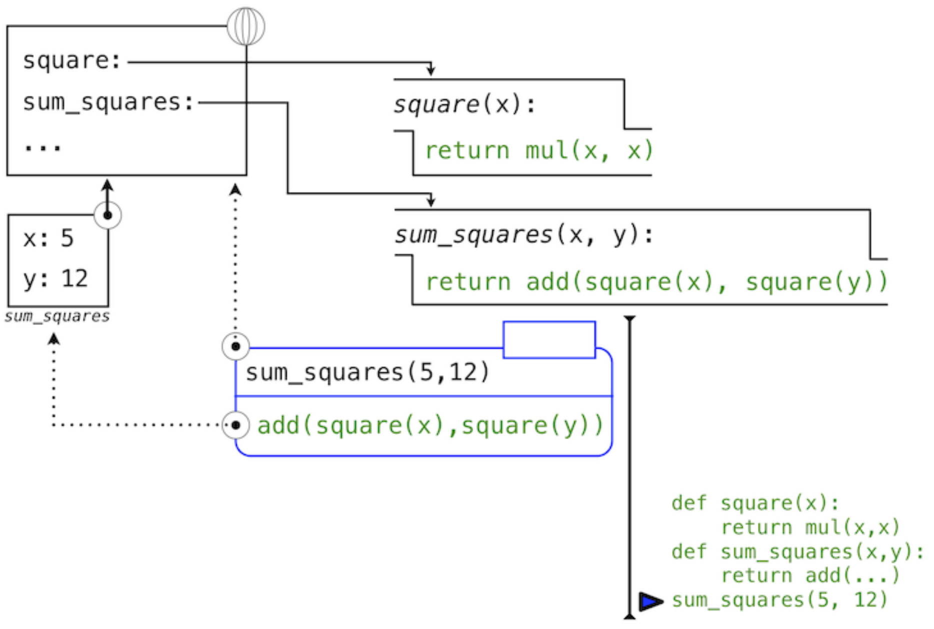


And the process that evaluates the following call expression:

```
>>> sum_squares(5, 12)
169
```

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.



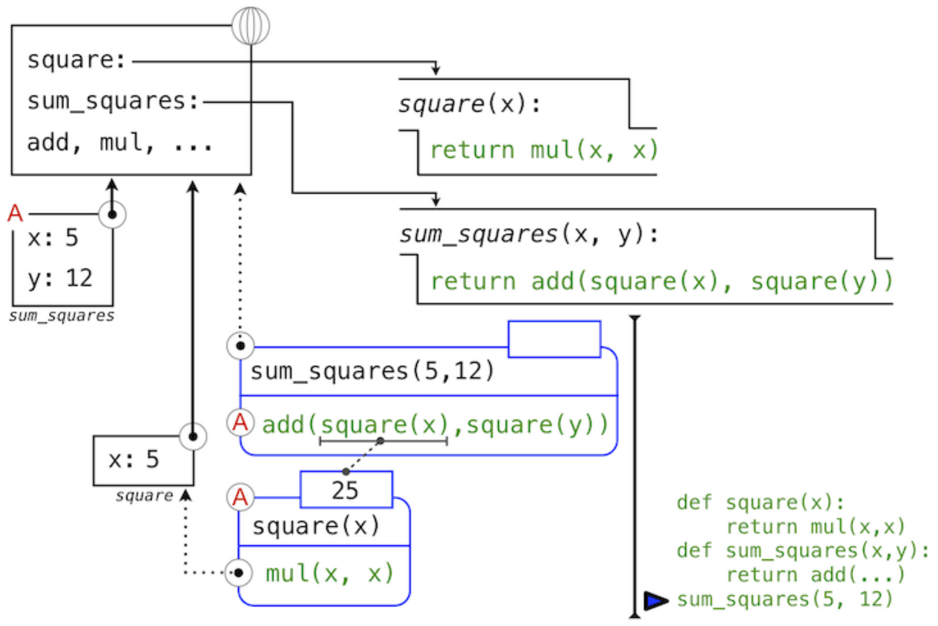
In this diagram, the local frame points to its successor, the global frame. All local frames must point to a predecessor, and these links define the sequence of frames that is the current environment.

The body of `sum_squares` contains this call expression:

```
      add      (  square(x)   ,  square(y)   )
    ________      __________     __________
  "operator"    "operand 0"    "operand 1"
```

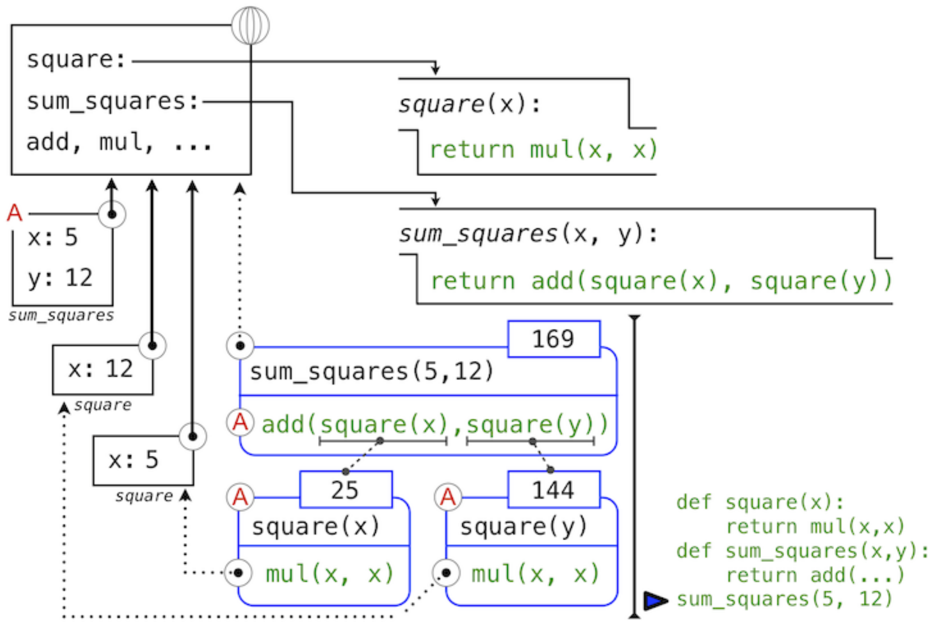
All three subexpressions are evalauted in the current environment, which begins with the frame labeled *sum_squares*. The operator subexpression `add` is a name found in the global frame, bound to the built-in function for addition. The two operand subexpressions must be evaluated in turn, before addition is applied. Both operands are evaluated in the current environment beginning with the frame labeled `sum_squares`. In the following environment diagrams, we will call this frame `A` and replace arrows pointing to this frame with the label `A` as well.

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.



Using this local frame, the body 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 environment frame that binds `x` to 12. Hence, `operand 1` evaluates to 144.



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

This figure, while complex, serves to illustrate many of the fundamental ideas we have developed so far. Names are bound to values, which spread across many local frames that all precede a single global frame that contains shared names. Expressions are tree-structured, and the environment must be augmented each time a subexpression contains a call to a user-defined function.

All of this machinery exists to ensure that names resolve to the correct values at the correct points in the expression tree. 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.

1.3.4 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. Our 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.

1.3.5 Practical Guidance: Choosing Names

The interchangeabily of names does not imply that formal parameter names do not matter at all. To 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 programming community. As a side effect of following these conventions, you will find that your code becomes more internally consistent.

1. Function names should be 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 should be 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 type of value that is allowed.
5. Single letter parameter names are acceptable when their role is obvious, but never use "l" (lowercase ell), "O" (capital oh), or "I" (capital i) to avoid confusion with numerals.

Review these guidelines periodically as you write programs, and soon your names will be delightfully Pythonic.

1.3.6 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.