Chapter 1
Hide contents

1.1 Getting Started

1.1.1 Programming in Python
1.1.2 Installing Python 3
1.1.3 Interactive Sessions
1.1.4 First Example
1.1.5 Errors

1.2 Elements of Programming

1.2.1 Expressions
1.2.2 Call Expressions
1.2.3 Importing Library Functions
1.2.4 Names and the Environment
1.2.5 Evaluating Nested Expressions
1.2.6 The Non-Pure Print Function

1.3 Defining New Functions

1.3.1 Environments
1.3.2 Calling User-Defined Functions
1.3.3 Example: Calling a User-Defined Function
1.3.4 Local Names
1.3.5 Choosing Names
1.3.6 Functions as Abstractions
1.3.7 Operators

1.4 Designing Functions

1.4.1 Documentation
1.4.2 Default Argument Values

1.5 Control

1.5.1 Statements
1.5.2 Compound Statements
1.5.3 Defining Functions II: Local Assignment
1.5.4 Conditional Statements
1.5.5 Iteration
1.5.6 Testing

1.6 Higher-Order Functions

1.6.1 Functions as Arguments
1.6.2 Functions as General Methods
1.6.3 Defining Functions III: Nested Definitions
1.6.4 Functions as Returned Values
1.6.5 Example: Newton's Method
1.6.6 Currying
1.6.7 Lambda Expressions

1.3 Defining New Functions

Video: Show Hide

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*.

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`.

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):
```

1.6.8 Abstractions and First-Class Functions
1.6.9 Function Decorators

1.7 Recursive Functions

1.7.1 The Anatomy of Recursive Functions
1.7.2 Mutual Recursion
1.7.3 Printing in Recursive Functions
1.7.4 Tree Recursion
1.7.5 Example: Partitions

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

1.3.1 Environments

Video: Show Hide

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.

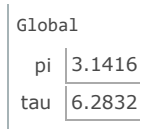
An environment in which an expression is evaluated consists of a sequence of *frames*, depicted as boxes. Each frame contains *bindings*, each of which associates a name with its corresponding value. There is a single *global* frame. Assignment and import statements add entries to the first frame of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

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

Edit code in Online Python Tutor

< Back End Forward >

line that has just executed
next line to execute



This *environment diagram* shows the bindings of the current environment, along with the values to which names are bound. The environment diagrams in this text are interactive: you can step through the lines of the small program on the left to see the state of the environment evolve on the right. You can also click on the "Edit code in Online Python Tutor" link to load the example into the Online Python Tutor, a tool created by Philip Guo for generating these environment diagrams. You are encouraged to create examples yourself and study the resulting environment diagrams.

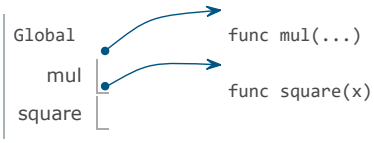
Functions appear in environment diagrams as well. An `import` statement binds a name to a built-in function. A `def` statement binds a name to a user-defined function created by the definition. The resulting environment after importing `mul` and defining `square` appears below:

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

Edit code in Online Python Tutor

< Back End Forward >

line that has just executed
next line to execute

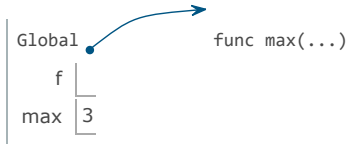


Each function is a line that starts with `func`, followed by the function name and formal parameters. Built-in functions such as `mul` do not have formal parameter names, and so `...` is always used instead.

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
```



Edit code in Online Python Tutor

< Back Step 3 of 4 Forward >

line that has just executed
next line to execute

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.

1.3.2 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:

1. Bind the arguments to the names of the function's formal parameters in a new *local* frame.
2. 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)
```

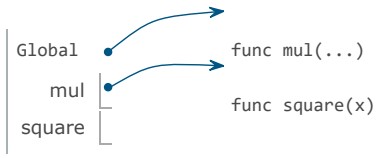
Edit code in Online Python Tutor

< Back Step 2 of 5 Forward >

line that has just executed
next line to execute

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)
```



Edit code in Online Python Tutor

< Back Step 3 of 5 Forward >

line that has just executed

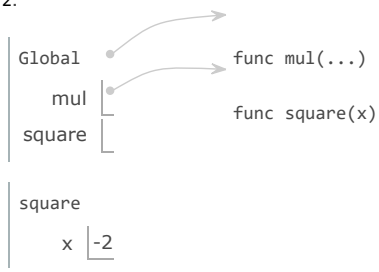
next line to execute

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)
```

Edit code in Online Python Tutor

< Back Step 4 of 5 Forward >



line that has just executed

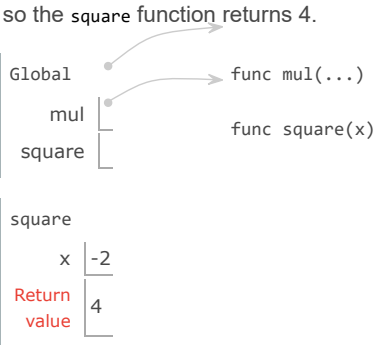
next line to execute

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)
```

Edit code in Online Python Tutor

< Back Step 5 of 5 Forward >



line that has just executed

next line to execute

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.

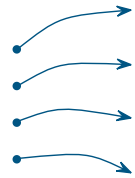
The order of frames in an environment affects the value returned by looking up a name in an expression. We stated previously that a name is evaluated to the value associated with that name in the current environment. We can now be more precise:

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.

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 will see how this model can serve as a blueprint for implementing a working interpreter for a programming language.

1.3.3 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.

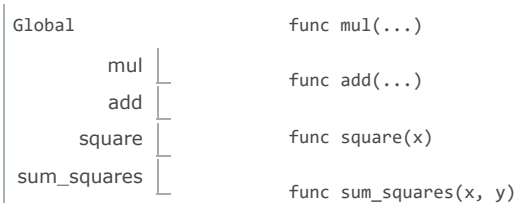


```
1  from operator import add, mul
2  def square(x):
3      return mul(x, x)
4
5  def sum_squares(x, y):
6      return add(square(x), square(y))
7
8  result = sum_squares(5, 12)
```

Edit code in Online Python Tutor

< Back Step 4 of 10 Forward >

line that has just executed
next line to execute



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.

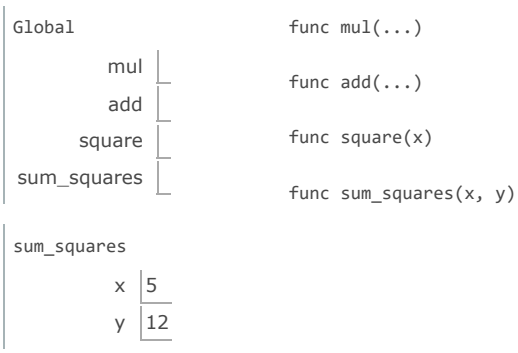
Next, Python applies `sum_squares`, which introduces a local frame that binds x to 5 and y to 12.

```
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)
```

Edit code in Online Python Tutor

< Back Step 5 of 10 Forward >

line that has just executed
next line to execute



The body of `sum_squares` contains this call expression:

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

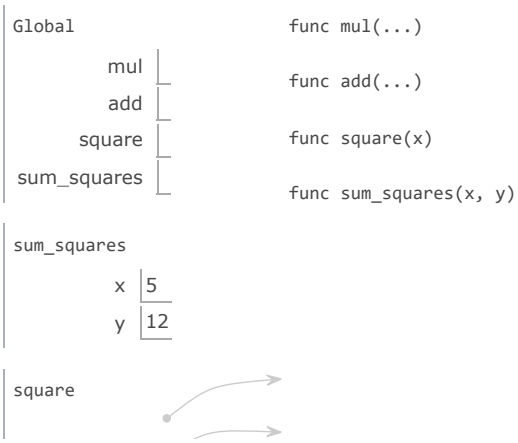
All three subexpressions are evaluated in the current environment, which begins with the frame labeled `sum_squares()`. The operator subexpression `add` is a name found in the global frame, bound to the built-in function for addition. The two operand subexpressions must be evaluated in turn, before addition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

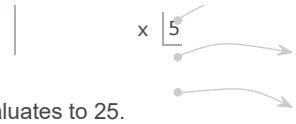
```
1  from operator import add, mul
2  def square(x):
3      return mul(x, x)
4
5  def sum_squares(x, y):
6      return add(square(x), square(y))
7
8  result = sum_squares(5, 12)
```

Edit code in Online Python Tutor

< Back Step 6 of 10 Forward >



line that has just executed
next line to execute



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

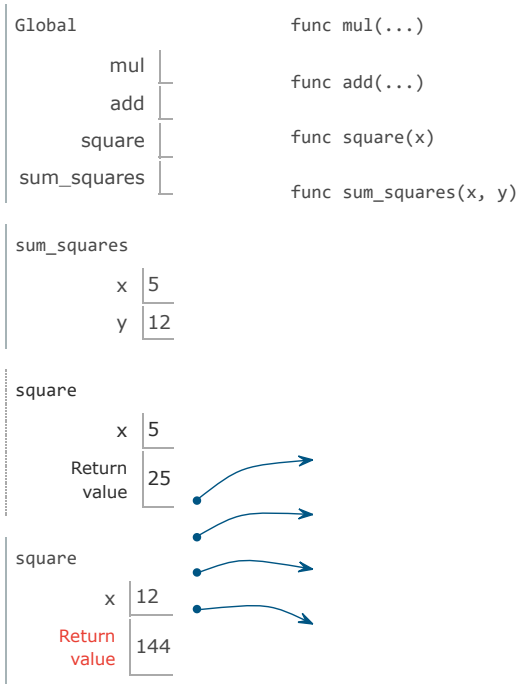
Our evaluation procedure now turns to `operand 1`, for which `y` names the number 12. Python evaluates the body of `square` again, this time introducing yet another local frame that binds `x` to 12. Hence, `operand 1` evaluates to 144.

```
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)
```

Edit code in Online Python Tutor

< Back Step 9 of 10 Forward >

line that has just executed
next line to execute



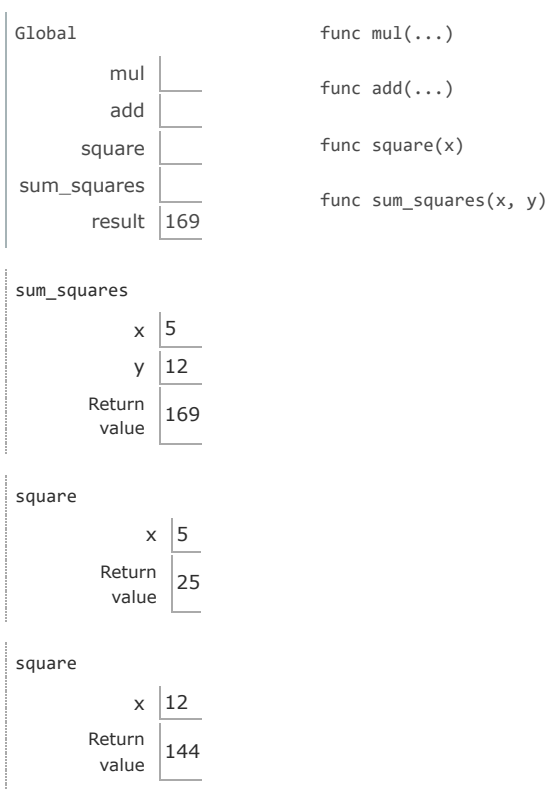
Finally, applying addition to the arguments 25 and 144 yields a final return value for `sum_squares`: 169.

```
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)
```

Edit code in Online Python Tutor

< Back End Forward >

line that has just executed
next line to execute



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.

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. 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.

1.3.5 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.

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.

```
>>> 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.

Aspects of a functional abstraction. To master the use of a functional abstraction, it is often useful to consider its three core attributes. The *domain* of a function is the set of arguments it can take. The *range* of a function is the set of values it can return. The *intent* of a function is the relationship it computes between inputs and output (as well as any side effects it might generate). Understanding functional abstractions via their domain, range, and intent is critical to using them correctly in a complex program.

For example, any `square` function that we use to implement `sum_squares` should have these attributes:

- The *domain* is any single real number.
- The *range* is any non-negative real number.
- The *intent* is that the output is the square of the input.

These attributes do not specify how the intent is carried out; that detail is abstracted away.

1.3.7 Operators

Video: Show Hide

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 short-hand 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.

Continue: 1.4 Designing Functions

Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.