INTRODUCTION TO

Programming in Python

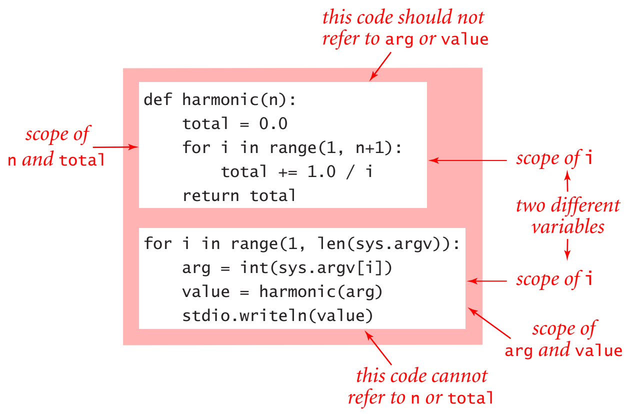
An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne • Robert Dondero

Introduction to Programming in Python

tion than a single number, boolean, or string. For example, you will see later in this section that you can use arrays as return values.

Scope. The *scope* of a variable is the set of statements that can refer to that variable directly. The scope of a function's local and parameter variables is limited to that function; the scope of a variable defined in global code—known as a *global variable*—is limited to the `.py` file containing that variable. Therefore, global code cannot refer to either a function's local or parameter variables. Nor can one function refer to either the local or parameter variables that are defined in another function. When a function defines a local (or parameter) variable with the same name as a global variable (such as `i` in PROGRAM 2.1.1), the variable name in the function refers to the local (or parameter) variable, not the global variable.



```
def harmonic(n):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total
```

```
for i in range(1, len(sys.argv)):
    arg = int(sys.argv[i])
    value = harmonic(arg)
    stdio.writeln(value)
```

Scope of local and parameter variables

A guiding principle when designing software is to define each variable so that its scope is as small as possible. One of the important reasons that we use functions is so that changes made to one part of a program will not affect an unrelated part of the program. So, while code in a function *can* refer to global variables, it *should not* do so: all communication from a caller to a function should take place via the function's parameter variables, and all communication from a function to its caller should take place via the function's return value. In SECTION 2.2, we consider a technique for removing most global code, thereby limiting scope and the potential for unexpected interactions.

Default arguments. A Python function may designate an argument to be *optional* by specifying a *default value* for that argument. If you omit an optional argument in a function call, then Python substitutes the default value for that argument. We have already encountered a few examples of this feature. For example, `math.log(x, b)` returns the base-`b` logarithm of `x`. If you omit the second argument, then `b` defaults to `math.e`—that is, `math.log(x)` returns the natural logarithm of `x`. It might appear that the `math` module has two different logarithm functions, but it actually has just one, with an optional argument and a default value.

You can specify an optional argument with a default value in a user-defined function by putting an equals sign followed by the default value after the parameter variable in the function signature. You can specify more than one optional argument in a function signature, but all of the optional arguments must follow all of the mandatory arguments.

For example, consider the problem of computing the nth *generalized harmonic number of order* r: $H_{n,\,r} = 1 + 1/2^r + 1/3^r + \ldots + 1/n^r$. For example, $H_{1,\,2} = 1$, $H_{2,\,2} = 5/4$, and $H_{2,\,2} = 49/36$. The generalized harmonic numbers are closely related to the Riemann zeta function from number theory. Note that the nth generalized harmonic number of order $r = 1$ is equal to the nth harmonic number. Therefore it is appropriate to use `1` as the default value for `r` if the caller omits the second argument. We specify by writing `r=1` in the signature:

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

With this definition, `harmonic(2, 2)` returns `1.25`, while both `harmonic(2, 1)` and `harmonic(2)` return `1.5`. To the client, it appears that we have two different functions, one with a single argument and one with two arguments, but we achieve this effect with a single implementation.

Side effects. In mathematics, a function maps one or more input values to some output value. In computer programming, many functions fit that same model: they accept one or more arguments, and their only purpose is to return a value. A *pure function* is a function that, given the same arguments, always return the same value, without producing any observable *side effects*, such as consuming input, producing output, or otherwise changing the state of the system. So far, in this section we have considered only pure functions.

However, in computer programming it is also useful to define functions that do produce side effects. In fact, we often define functions whose only purpose is to produce side effects. An explicit `return` statement is optional in such a function: control returns to the caller after Python executes the function's last statement. Functions with no specified return value actually return the special value `None`, which is usually ignored.

For example, the `stdio.write()` function has the side effect of writing the given argument to standard output (and has no specified return value). Similarly, the following function has the side effect of drawing a triangle to standard drawing (and has no specified return value):

```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

It is generally poor style to compose a function that both produces side effects and returns a value. One notable exception arises in functions that read input. For example, the `stdio.readInt()` function both returns a value (an integer) and produces a side effect (consuming one integer from standard input).

Type checking. In mathematics, the definition of a function specifies both the domain and the range. For example, for the harmonic numbers, the domain is the positive integers and the range is the positive real numbers. In Python, we do not specify the types of the parameter variables or the type of the return value. As long as Python can apply all of the operations within a function, Python executes the function and returns a value.

If Python cannot apply an operation to a given object because it is of the wrong type, it raises a run-time error to indicate the invalid type. For example, if you call the `square()` function defined earlier with an `int` argument, the result is an `int`; if you call it with a `float` argument, the result is a `float`. However, if you call it with a string argument, then Python raises a `TypeError` at run time.

This flexibility is a popular feature of Python (known as *polymorphism*) because it allows us to define a single function for use with objects of different types. It can also lead to unexpected errors when we call a function with arguments of unanticipated types. In principle, we could include code to check for such errors, and we could carefully specify which types of data each function is supposed to work with. Like most Python programmers, we refrain from doing so. However, in this book, our message is that *you should always be aware of the type of your data*, and the functions that we consider in this book are built in line with this philosophy, which admittedly clashes with Python's tendency toward polymorphism. We will discuss this issue in some detail in SECTION 3.3.

The table below summarizes our discussion by collecting together the function definitions that we have examined so far. To check your understanding, take the time to reread these examples carefully.

primality test

```
def isPrime(n):
    if n < 2: return False
    i = 2
    while i*i <= n:
        if n % i == 0: return False
        i += 1
    return True
```

hypotenuse of a right triangle

```
def hypot(a, b)
    return math.sqrt(a*a + b*b)
```

generalized harmonic number

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

draw a triangle

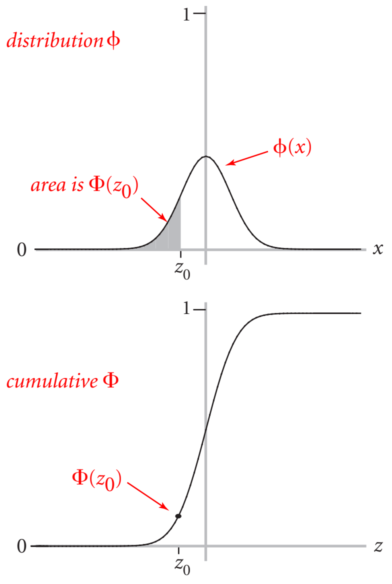
```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

Typical code for implementing functions

Implementing mathematical functions Why not just use the Python built-in functions and those that are defined in the standard or extension Python modules? For example, why not use the `math.hypot()` function instead of defining our own `hypot()` function? The answer to this question is that we *do* use such functions when they are present (because they are likely to be faster and more accurate). However, there is an unlimited number of functions that we may wish to use and only a finite number of functions is defined in the Python standard and extension modules. When you need a function that is not defined in the Python standard or extension modules, you need to define the function yourself.

As an example, we consider the kind of code required for a familiar and important application that is of interest to many potential college students in the United States. In a recent year, over 1 million students took the Scholastic Aptitude Test (SAT). The test consists of two major sections: critical reading and mathematics. Scores range from 200 (lowest) to 800 (highest) on each section, so overall test scores range from 400 to 1600. Many universities consider these scores when making important decisions. For example, student athletes are required by the National Collegiate Athletic Association (NCAA), and thus by many universities, to have a combined score of at least 820 (out of 1600), and the minimum eligibility requirement for certain academic scholarships is 1500 (out of 1600). What percentage of test takers is ineligible for athletics? What percentage is eligible for the scholarships?

Two functions from statistics enable us to compute accurate answers to these questions. The *standard normal (Gaussian) probability density function* is characterized by the familiar bell-shaped curve and defined by the formula $\phi(x) = e^{-x^2/2}/\sqrt{2\pi}$. The standard normal (Gaussian) *cumulative distribution function* $\Phi(z)$ is defined to be the area under the curve defined by $\phi(x)$ above the x-axis and to the left of the vertical line $x=z$. These functions play an important role in science, engineering, and finance because they arise as accurate models throughout the natural world and because they are essential in understanding experimental error. In particular, these functions are known to accurately describe the distribution of test scores in our example, as a function of the mean (average value of the scores) and the standard deviation (square root of the average of the squares of the differences between each score and the mean), which are published each year. Given the mean μ and the standard deviation σ of the test scores, the percentage of students with scores less than a given value z is closely approximated by the function $\Phi(z, \mu, \sigma) = \Phi((z-\mu)/\sigma)$. Functions to calculate ϕ and Φ are not available in Python's `math` module, so we develop our own implementations.



Gaussian probability functions

Closed form. In the simplest situation, we have a closed-form mathematical formula defining our function in terms of functions that are implemented in Python's `math` module. This situation is the case for ϕ—the `math` module includes functions to compute the exponential and the square root functions (and a constant value for π), so a function `pdf()` corresponding to the mathematical definition is easy to implement. For convenience, `gauss.py` (Program 2.1.2) uses the default arguments $\mu = 0$ and $\sigma = 1$ and actually computes $\phi(x, \mu, \sigma) = \phi((x-\mu) / \sigma) / \sigma$.

No closed form. If no formula is known, we may need a more complicated algorithm to compute function values. This situation is the case for Φ—no closed-form expression exists for this function. Algorithms to compute function values sometimes follow immediately from Taylor series approximations, but developing reliably accurate implementations of mathematical functions is an art and a science that needs to be addressed carefully, taking advantage of the knowledge built up in mathematics over the past several centuries. Many different approaches have been studied for evaluating Φ. For example, a Taylor series approximation to the ratio of Φ and ϕ turns out to be an effective basis for evaluating the function:

$$\Phi(z) = 1/2 + \phi(z)\,(z + z^3/3 + z^5/(3\cdot5) + z^7/(3\cdot5\cdot7) + \ldots).$$

This formula readily translates to the Python code for the function `cdf()` in Program 2.1.2. For small (respectively large) z, the value is extremely close to 0 (respectively 1), so the code directly returns 0 (respectively 1); otherwise, it uses the Taylor series to add terms until the sum converges. Again, for convenience, Program 2.1.2 actually computes $\Phi(z, \mu, \sigma) = \Phi((z-\mu) / \sigma)$, using the defaults $\mu = 0$ and $\sigma = 1$.

Running `gauss.py` with the appropriate arguments on the command line tells us that about 17% of the test takers were ineligible for athletics in a year when the mean was 1019 and the standard deviation was 209. In the same year, about 1% percent qualified for academic scholarships.

Computing with mathematical functions of all sorts plays a central role in science and engineering. In a great many applications, the functions that you need are expressed in terms of the functions in Python's `math` module, as we have just seen with `pdf()`, or in terms of a Taylor series approximation or some other formulation that is easy to compute, as we have just seen with `cdf()`. Indeed, support for such computations has played a central role throughout the evolution of computing systems and programming languages.