1.6 Higher-Order Functions

composingprograms.com/pages/16-higher-order-functions.html

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.

1.6.1 Functions as Arguments

Consider the following three functions, which all compute summations. The first, `sum_naturals`, computes the sum of natural numbers up to `n`:

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

>>> sum_naturals(100)
5050
```

The second, `sum_cubes`, computes the sum of the cubes of natural numbers up to `n`.

```
>>> def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k*k*k, k + 1
        return total

>>> sum_cubes(100)
25502500
```

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

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

which converges to pi very slowly.

```
>>> def pi_sum(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + 8 / ((4*k-3) * (4*k-1)), k + 1
        return total

>>> pi_sum(100)
3.1365926848388144
```

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, k = 0, 1
    while k <= n:
        total, k = total + <term>(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 names `term` ensures that the result `1*1*1 + 2*2*2 + 3*3*3 = 36` is computed correctly. In this example, frames which are no longer needed are removed to save space.

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, k = 0, 1
        while k <= n:
            total, k = total + term(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 defininining 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
```

1.6.2 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. 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
```

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
>>> def improve_test():
        approx_phi = improve(golden_update, square_close_to_successor)
        assert approx_eq(phi, approx_phi), 'phi differs from its approximation'

>>> improve_test()
```

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

1.6.3 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.

1. Each user-defined function has a parent environment: the environment in which it was defined.
2. 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`.

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 realize 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*.

1.6.4 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.

1.6.5 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 optimization.

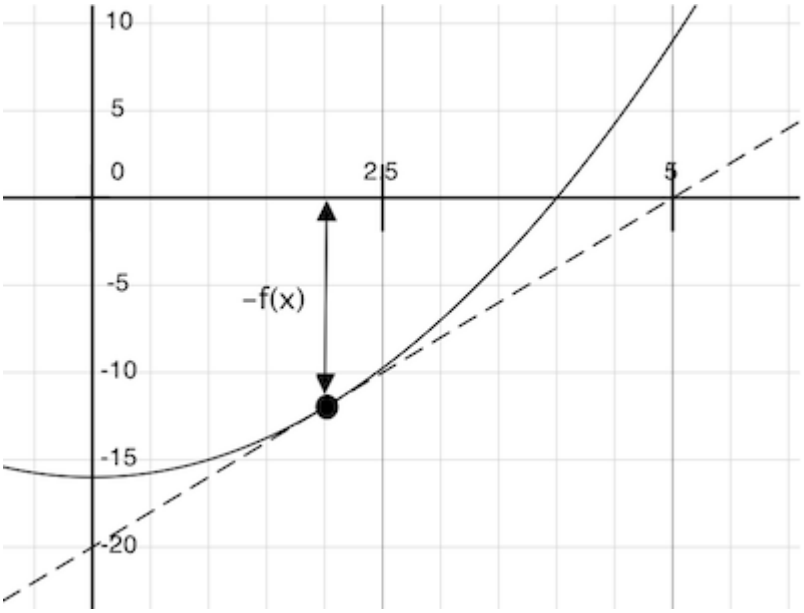
Newton's method is a classic iterative approach to finding the arguments of a mathematical function that yield a return value of 0. These values are called the *zeros* of the function. Finding a zero of a function is often equivalent to solving some other problem of interest, such as computing a square 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))(x,f(x)) that has the same slope as the curve for function f(x)f(x) at that point. Such a line is called the *tangent*, and its slope is called the *derivative* of ff at xx.

This line's slope is the ratio of the change in function value to the change in function argument. Hence, translating xx by f(x)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` algorithm, and a comparison to see if f(x)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 nn. The degree nn root of aa is xx such that x·x·x…x=ax·x·x…x=a with xx repeated nn times. For example,

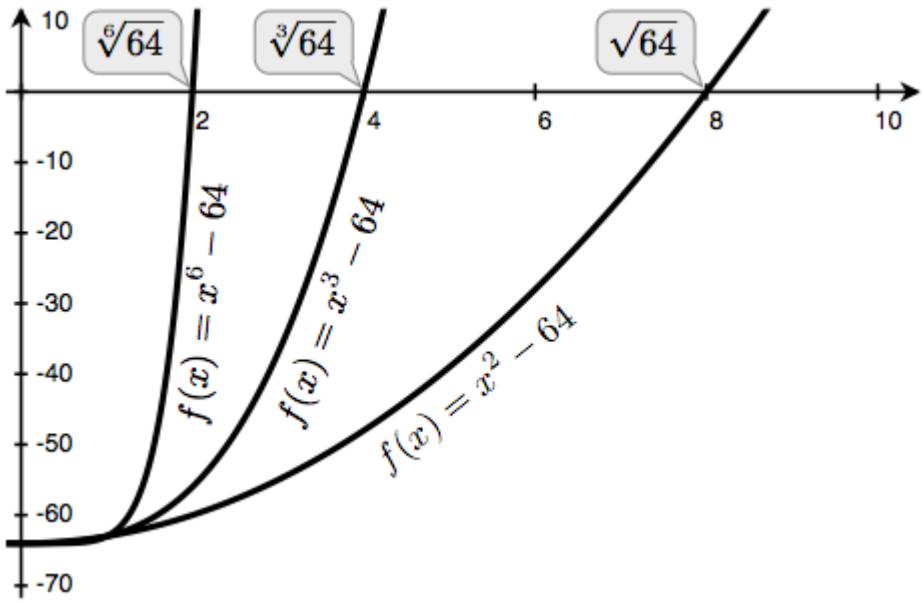
- The square (second) root of 64 is 8, because 8·8=648·8=64.
- The cube (third) root of 64 is 4, because 4·4·4=644·4·4=64.
- The sixth root of 64 is 2, because 2·2·2·2·2·2=642·2·2·2·2·2=64.

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

- The square root of 64 (written 64−−√64) is the value xx such that x2−64=0x2−64=0

- More generally, the degree nn root of aa (written a−−√nan) is the value xx such that xn−a=0xn−a=0

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



Video: Show Hide

We first implement `square_root` by defining `f` and its derivative `df`. We use from calculus the fact that the derivative of f(x)=x2−af(x)=x2−a is the linear function df(x)=2xdf(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)

>>> square_root_newton(64)
8.0
```

Generalizing to roots of arbitrary degree nn, we compute f(x)=xn−af(x)=xn−a and its derivative df(x)=n·xn−1df(x)=n·xn−1.

```
>>> def power(x, n):
        """Return x * x * x * ... * x for x repeated n times."""
        product, k = 1, 0
        while k < n:
            product, k = product * x, k + 1
        return product
```

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

>>> nth_root_of_a(2, 64)
8.0
>>> nth_root_of_a(3, 64)
4.0
>>> 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.

1.6.6 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

>>> 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
```

1.6.7 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.

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 favor.

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 "ŷ . y × y". But frustrated typographers moved the hat to the left of the parameter and changed it to a capital lambda: "Λy . y × y"; from there the capital lambda was changed to lowercase, and now we see "λy . y × 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. We will revisit this topic when we study the design of interpreters in Chapter 3.

1.6.8 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 generalize 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:

1. They may be bound to names.
2. They may be passed as arguments to functions.
3. They may be returned as the results of functions.
4. They may be included in data structures.

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

1.6.9 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)
```

In the projects associated with this text, decorators are used for tracing, as well as selecting which functions to call when a program is run from the command line.

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. A short tutorial on decorators by Ariel Ortiz gives further examples for interested students.