CS1010X: Programming Methodology I
Chapter 6: Higher-Order Functions

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 6: Higher-Order Functions

Learning Objectives

By the end of this chapter, *students will be able to*:

- **Explain** what it means for functions to be *first-class* values in Python, and use them as arguments and return values.
- **Refactor** families of similar functions (e.g. various sums and products) into reusable higher-order functions that capture common patterns.
- **Construct** anonymous functions with `lambda` and apply them appropriately for short, one-off behaviours.
- **Apply** higher-order functions to numerical problems: definite integration, numerical differentiation, and root finding (Newton's method).
- **Generalise** recursive computations with `fold` and `fold2` and connect them to familiar operations such as sum, product and list-processing patterns.
- **Reason about** function composition and predict how composing higher-order functions changes the number of times a function is applied.

6.1 Higher-Order Functions as Abstractions

We begin with a simple observation: as programs grow, we often write families of functions that look almost the same. This is a sign that a useful abstraction is missing.

6.1.1 Summing Values

Consider the three different mathematical sums below:

Summing integers. Suppose we want to calculate the sum of integers from a to b:

$$\sum_{n=a}^{b} n = a + (a+1) + \cdots + b.$$

```python
def sum_integers(a, b):
    if a > b:
        return 0
    else:
        return a + sum_integers(a + 1, b)
```

Summing cubes. Now, suppose we want to calculate the sum of cubes from a to b:

$$\sum_{n=a}^{b} n^3 = a^3 + (a+1)^3 + \cdots + b^3.$$

```python
def sum_cubes(a, b):
    if a > b:
        return 0
    else:
        return (a * a * a) + sum_cubes(a + 1, b)
```

Approximating π. Finally, consider the calculation for $\frac{\pi}{8}$ using the series below:

$$\frac{\pi}{8} = \frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \dots$$

```python
def pi_sum(a, b):
    if a > b:
        return 0
    else:
        term = 1 / (a * (a + 2))
        return term + pi_sum(a + 4, b)
```

All three procedures:

1. test a **base case** (`a > b`),
2. compute a **term** from the current index (`a`, `a*a*a`, or $\frac{1}{a(a+2)}$),
3. move to the **next** index (add 1 or 4),
4. add the term to the recursive result.

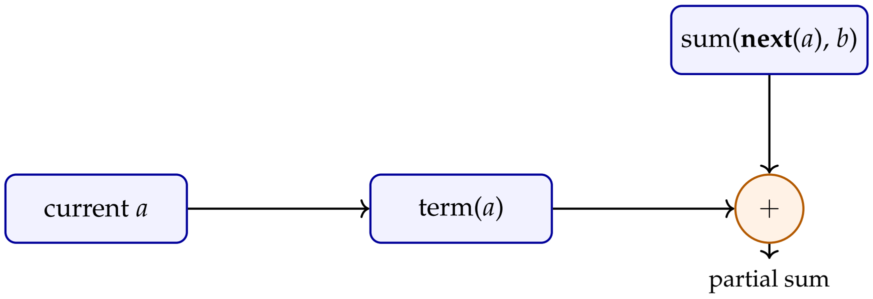


Figure 6.1: The common structure behind `sum_integers`, `sum_cubes`, and `pi_sum`.

Which parts change?
• `term`: how to transform a
• `next`: how to move to the next index

We would like a *single* piece of code that captures this structure and receives the "changing parts" as parameters.

6.1.2 First-Class Functions and sums

Python treats functions as **first-class values**:

- they can be assigned to variables,
- passed as arguments,
- returned as results,
- stored in data structures.

This allows us to write "functions that consume functions":

```python
def sum(term, a, next, b):
    """Return term(a) + term(next(a)) + ... up to b."""
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)
```

Our three earlier functions become thin wrappers of our new function:

```python
def sum_integers(a, b):
    return sum(lambda x: x,
                         a,
                         lambda x: x + 1,
                         b)

def sum_cubes(a, b):
    return sum(lambda x: x * x * x,
                         a,
                         lambda x: x + 1,
                         b)

def pi_sum(a, b):
    return sum(lambda x: 1.0 / (x * (x + 2)),
                         a,
                         lambda x: x + 4,
                         b)
```

Passing a Function vs Calling It

```python
sum(f, 1, next, 10)   # passes the function f
sum(f(), 1, next, 10) # calls f *once* and passes its result
```

In a higher-order context, you almost always want the first form. So, make sure to not accidentally write the second form if you mean the first.

6.1.3 Review: Lambdas

Lambda expressions are anonymous, one-line functions:

```
lambda <parameters>: <expression>
```

```
lambda x: x * x          # square
lambda x, y: x + 2 * y   # two-argument function
```

Lambda expressions allow us to succinctly write one-off functions, which is useful when passing them in as parameters.

6.1.4 Products and Folds

The same recursive pattern works for products: only the combiner and base case change.

```
def product(term, a, next, b):
    """Return term(a) * term(next(a)) * ... up to b."""
    if a > b:
        return 1
    else:
        return term(a) * product(term, next(a), next, b)
```

For example:

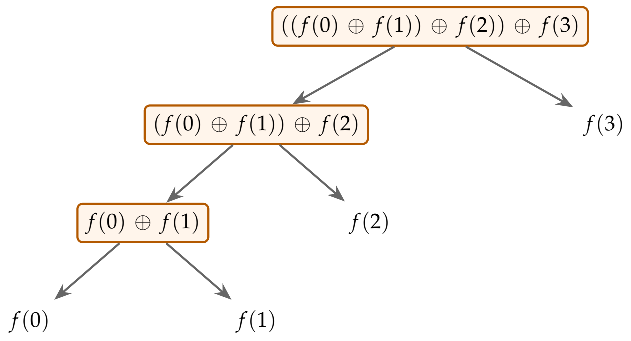
```
def factorial(n):
    return product(lambda x: x, 1, lambda x: x + 1, n)
```

Now, it is inconvenient to have to define a new function every time we want to change the operation. What if we wanted to take the bitwise XOR of all numbers? We can abstract out the function used to combine values. We term this function `fold`:

```
def fold(op, f, n):
    """Compute f(0) op f(1) op ... op f(n)."""
    if n == 0:
        return f(0)
    else:
        return op(f(n), fold(op, f, n - 1))
```

`fold` can implement many operations, e.g. exponentiation:

```
def expt(a, n):
    return fold(lambda x, y: x * y,    # op
                lambda k: a,           # f ignores k
                n - 1)                 # indices 0..n-1
```



$((f(0) \oplus f(1)) \oplus f(2)) \oplus f(3)$
$(f(0) \oplus f(1)) \oplus f(2)$
$f(3)$
$f(0) \oplus f(1)$
$f(2)$
$f(0)$
$f(1)$

Figure 6.2: Left fold of $f(0), f(1), f(2), f(3)$ with a binary operator $\oplus$.

Left vs Right Fold, Associativity and Commutativity

Our `fold` is a **right fold**:

$$\text{fold}(op, f, n) = f(n) \ \oplus \ \big(f(n-1) \ \oplus \ \big(\ldots \ \oplus \ f(0)\big)\ldots\big),$$

where $\oplus$ stands for `op`.
A **left fold** would combine in the opposite direction:

$$(((f(n) \ \oplus \ f(n-1)) \ \oplus \ f(n-2)) \ \oplus \ \ldots \ \oplus \ f(0)).$$

Associativity. An operator $\oplus$ is *associative* if

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \quad \text{for all } a, b, c.$$

If $\oplus$ is associative, then left and right folds over the *same* sequence give the same result.

Commutativity. An operator is *commutative* if

$$a \oplus b = b \oplus a.$$

Commutativity is about *swapping* arguments; associativity is about *regrouping* them. Both folds differ in regrouping, so equality depends only on **associativity**.

When will left and right fold give the same result?
1. Same result (associative operators).

- Integer addition:

$$((1+2)+3) = 3+3 = 6, \quad 1+(2+3) = 1+5 = 6.$$

  So for the list $(1,2,3)$, both left and right folds with $\oplus = +$ give 6.

2. Different result (non-associative operators).

- Subtraction:

$$((1-2)-3) = (-1)-3 = -4, \quad 1-(2-3) = 1-(-1) = 2.$$

  So left fold of $(1,2,3)$ with $\oplus = -$ gives -4, but right fold gives 2.

6.1.5 Extending Fold to Arbitrary Ranges

Now, let us further generalise fold. Suppose we want to apply fold on a defined intervals $[a, b]$ instead of $[0, n]$. Furthermore, we may want to be able to increase the steps by more than 1. How can we do that instead? We can define another function `fold2` which takes in the starting and ending indices, as well as the `next` function, which determines how the next index in the sequence is computed.

```python
def fold2(op, term, a, next, b, base):
    """General fold over a range a, next(a), ..., up to b."""
    if a > b:
        return base
    else:
        return op(term(a),
                  fold2(op, term, next(a), next, b, base))
```

Then, we can write our previous functions in terms of this new `fold2` function.

```python
def sum(term, a, next, b):
    return fold2(lambda x, y: x + y, term, a, next, b, 0)

def product(term, a, next, b):
    return fold2(lambda x, y: x * y, term, a, next, b, 1)
```

Looking Forward: Parallelising Folds

As you will see in CS2030/S and higher-level modules, parallelization is a powerful technique used in computing today.

In particular, the tree-shaped structure of a fold suggests how to *parallelise* computations on large data.

- If the operator $\oplus$ is **associative** (e.g. $+$, $\times$, max, min), we can safely regroup terms:

$$(((x_0 \oplus x_1) \oplus x_2) \oplus x_3) = (x_0 \oplus x_1) \oplus (x_2 \oplus x_3) = x_0 \oplus (x_1 \oplus (x_2 \oplus x_3)).$$

- This means we can:
  1. split the range $[a, b]$ into chunks,
  2. run `fold2` on each chunk *in parallel*,
  3. combine the partial results with the same operator $\oplus$.

  This is essentially the idea behind **parallel reduction / folds**.
- Commutativity ($x \oplus y = y \oplus x$) is **not** required for correctness, but if the operation is also commutative then the final result does not depend on how we split or reorder the chunks.

In this module you will not need to implement parallel folds, but it is useful to remember: *a good abstraction like `fold2` also makes parallel and distributed implementations much easier.*

6.2 Standard Sequence Patterns: `map`, `filter`, `accumulate`

Many exam questions are built from a small library of sequence-processing functions (including the sum, product and fold operations above). We shall analyse what each of them do.

In general, these functions can be written to take in any ordered sequence, but we will use tuples in most cases. We will learn more about tuples in future chapters, but at a high level, tuples are simply an ordered sequence of values, similar to strings. We can index and slice them similarly to strings as well.

```python
def enumerate_interval(low, high):
    """Return (low, low+1, ..., high)."""
    return tuple(range(low, high + 1))

def map(fn, seq):
    """Apply fn to every element of seq."""
    if seq == ():
        return ()
    else:
        return (fn(seq[0]),) + map(fn, seq[1:])

def filter(pred, seq):
    """Keep elements x in seq where pred(x) is True."""
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],) + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])

def accumulate(fn, initial, seq):
    """Right fold of seq with binary function fn and base initial."""
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))
```

6.2.1 Map: transform every element.

$$\text{map}(fn, (x_0, \ldots, x_{n-1})) = (fn(x_0), \ldots, fn(x_{n-1})).$$

Example:

```python
map(lambda x: x * x, (1, 2, 3)) # => (1, 4, 9)
```

In the above, what happens in each step is that function is repeatedly applied to the head element, and `map` is applied to the rest of the tuple.

$$\text{map}(\text{fn}, (1, 2, 3)) = (\text{fn}(1),) + \text{map}(\text{fn}, (2, 3)).$$

Therefore, map can be viewed as having:

$$\text{head} = \texttt{seq[0]}, \quad \text{tail} = \texttt{seq[1:]}, \quad \text{result} = (\text{fn}(\text{head}),) + \text{map}(\text{fn}, \text{tail}).$$

6.2.2 Filter: select some elements.

$$\text{filter}(\text{pred}, (x_0, \ldots, x_{n-1})) = (x_i \mid \text{pred}(x_i) \text{ is True, in order}).$$

Example:

```python
filter(lambda x: x % 2 == 0, (1, 2, 3, 4))
    # => (2, 4)
```

In the above, what happens is that every element that fulfils the predicate is returned, while elements not fulfilling the predicate are discarded. Therefore,

$$\text{filter}(\text{pred}, (h, \text{tail})) = \begin{cases} (h,) + \text{filter}(\text{pred}, \text{tail}) & \text{if pred}(h) \\ \text{filter}(\text{pred}, \text{tail}) & \text{otherwise.} \end{cases}$$

6.2.3 Accumulate: fold a sequence.

$$\text{accumulate}(fn, initial, (x_0, \ldots, x_{n-1})) = x_0 \otimes (x_1 \otimes \ldots (x_{n-1} \otimes initial) \ldots),$$

where $\otimes$ stands for the binary operator `fn`. It is a *right fold*.

This is very similar to the right fold above.

6.2.4 Chaining Operations

Notice that every operation above returns a sequence. Thus, we can chain these sequences.

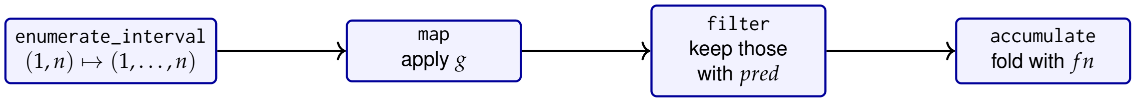


Figure 6.3: An example pipeline: generate a range, transform, filter, then fold.

Exam Tip: Reading Nested Patterns

For an expression such as

```
accumulate(fn, 0, map(g, enumerate_interval(1, n)))
```

always work from the inside out:

1. **Sequence**: `enumerate_interval(1, n)` gives $(1, 2, \ldots, n)$.
2. **Transformation**: `map(g, ...)` produces the sequence $(g(1), g(2), \ldots, g(n))$.
3. **Combination**: `accumulate(fn, 0, ...)` folds that sequence with `fn`, starting from `0`.

In more complicated examples, identify these three pieces first: *what sequence? what per-element operation? what final fold?* Once this is clear, the algebra is usually straightforward.

6.3 Numerical Methods as Higher-Order Patterns

The same abstractions are powerful tools in numerical analysis. We will also see how we can chain functions to form new functions.

6.3.1 Numerical Integration

Using the midpoint rule, we can mathematically approximate $\int_a^b f(x)\,dx$ by doing a Riemann sum. The smaller the value of Δx, the more accurate the approximation:

$$\int_a^b f(x)\,dx \;\approx\; \sum_{k=0}^{N-1} f\left(a + \left(k + \tfrac{1}{2}\right)\Delta x\right) \cdot \Delta x.$$

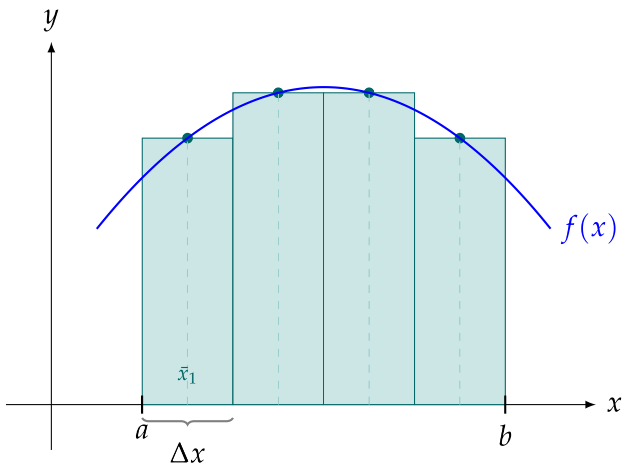


Figure 6.4: Approximating $\int_a^b f(x)\,dx$ via midpoint Riemann sums.

Using sum:

```python
def integral(f, a, b, dx):
    """Approximate the integral of f from a to b using step size dx.
        """
    def add_dx(x):
        return x + dx

    return dx * sum(f,
                    a + dx / 2,
                    add_dx,
                    b)
```

Now, computing the integral is a simple application of the sum function.

6.3.2 Derivatives as Higher-Order Functions

We approximate the derivative of g using a small step δ:

$$\frac{d}{dx}g(x) \approx \frac{g(x+\delta)-g(x)}{\delta}.$$

```python
def deriv(g):
    """Return a function that approximates the derivative of g."""
    dx = 0.00001
    return lambda x: (g(x + dx) - g(x)) / dx
```

`deriv` itself returns a new function, which can then be operated on or called.

Higher-Order Derivatives

Because `deriv` *returns a function*, we can apply it *again* to obtain higher-order derivatives.

Second derivative. Given a function g, its (approximate) second derivative is

$$g''(x) \approx \frac{d}{dx}\big(g'(x)\big).$$

In code:

```python
def second_deriv(g):
    return deriv(deriv(g))
```

Usage example:

```python
from math import sin, pi

g_prime  = deriv(sin)           # approx cos
g_second = second_deriv(sin)    # approx -sin

print(g_second(pi / 2))         # ~ 0 (since -sin(pi/2) = -1)
```

You can similarly obtain third and higher derivatives by composing `deriv` with itself multiple times, e.g. `deriv(deriv(deriv(g)))`.

6.3.3 Newton's Method

Given $g(x)$, we would like to find a root of the function, that is, we seek x^* with $g(x^*) = 0$. We can solve this numerically using Newton's Method, which iteratively finds closer and closer values to the root of the equation.

$$x_{\text{new}} = x - \frac{g(x)}{\left(\frac{d}{dx}g(x)\right)}.$$

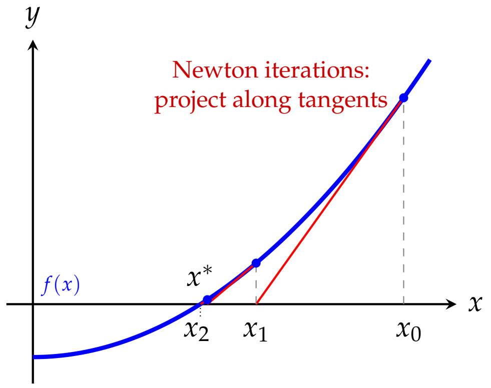


Figure 6.5: Newton's method for $f(x) = 0.1x^2 - 1$: successive tangent iterations $x_0 \to x_1 \to x_2 \to x_3$ converge to the root $x^* \approx \sqrt{10}$.

We can implement this using the derivative function we wrote above. Note that by doing this, we are treating derivative as a new abstraction that we can use.

```python
def newtons_method(g, first_guess):
    """Return an approximate root of g using Newton's method."""
    Dg = deriv(g)

    def new_guess(x):
        return x - g(x) / Dg(x)

    def is_close_enough(v):
        return abs(v) < 0.0001

    def attempt(guess):
        if is_close_enough(g(guess)):
            return guess
        else:
            return attempt(new_guess(guess))
    return attempt(first_guess)
```

For example, $\sqrt{a}$ is a root of $x^2 - a$. In effect, we have now used the abstraction of Newton's Method to find the square root of any number, and each of the functions above are flexible enough to be used for different purposes.

```python
def sqrt(a):
    return newtons_method(lambda x: x * x - a, a / 2)
```

6.4 Function Composition

We now turn to a different but related theme: *functions that manipulate functions* through composition. We have previously made use of functions to build other, more complicated functions, but we can also operate on functions directly.

6.4.1 Basic Combinators

Consider the definition of `compose`, `twice` and `thrice`. We can then compose these functions to form new functions.

```python
def compose(f, g):
    return lambda x: f(g(x))

def twice(f):
    return lambda x: f(f(x))

def thrice(f):
    return lambda x: f(f(f(x)))
```

Let `add1(x)` be `x + 1`. Then:

- `twice(add1)` is $x \mapsto x + 2$,
- `thrice(add1)` is $x \mapsto x + 3$,
- `compose(twice(add1), thrice(add1))` is $x \mapsto x + 5$.

Parentheses

Python application is left-associative:

$$\texttt{twice(thrice)(f)(0)} = (((\texttt{twice(thrice)})(f))(0)).$$

Always add parentheses and label each intermediate expression with its "type": number, 1-argument function, 2-argument function, etc.

6.4.2 Using Combinators

In many exam questions, the base function f is something simple like "add 1", and the main task is to count how many times it is applied.

Importantly, if we know how many times the base function f is applied, even if we change the function to "add 2" instead, we can easily find the new answer.

For each of the functions below, we apply some shorthands to quickly determine the answer, but it is **strongly recommended** to sit down with pen and paper and trace through the lambda expansions to ensure you understand why they work as is. It is important to build intuition for how these functions work, because more complicated variants often appear in examinations.

Powers of a Function. We write f^k to denote "apply f exactly k times".

If $g = f^k$ then:

- `twice(g)` applies g twice, so the exponent becomes $2k$,
- `thrice(g)` applies g three times, so the exponent becomes $3k$.

Example 1: `twice(thrice)(f)`

$$\texttt{twice(thrice)(f)} = \texttt{thrice}(\texttt{thrice}(f)).$$

Start with f^1. Each `thrice` multiplies the exponent by 3:

$$f^1 \xrightarrow{\texttt{thrice}} f^3 \xrightarrow{\texttt{thrice}} f^9.$$

So `twice(thrice)(f)` is f^9. If `f(x)=x+1`, then `twice(thrice)(f)(0) = 9`.

Example 2: `thrice(twice)(f)`

$$\texttt{thrice(twice)(f)} = \texttt{twice}(\texttt{twice}(\texttt{twice}(f))).$$

Now each `twice` doubles the exponent:

$$f^1 \xrightarrow{\texttt{twice}} f^2 \xrightarrow{\texttt{twice}} f^4 \xrightarrow{\texttt{twice}} f^8.$$

So `thrice(twice)(f)` is f^8, and `thrice(twice)(f)(0) = 8` if `f(x)=x+1`.

Quick Summary

For expressions built from `once`, `twice`, `thrice`, etc.:

1. Treat the base function f as f^1.
2. Each combinator multiplies the exponent (`twice`: $\times 2$, `thrice`: $\times 3$).
3. Follow the chain of combinators step by step, updating the exponent.

This is usually faster than manually expanding all lambdas.

6.4.3 Chaining Combinators

The key observation is that `twice` and `thrice` themselves are functions that take a *unary function* and return a unary function. So they can be passed into each other just like f.

Recall:

$$\texttt{twice}(f) = f^2, \qquad \texttt{thrice}(f) = f^3.$$

Example 3: `twice(twice)(f)` First, parenthesise:

$$\texttt{twice(twice)(f)} = \big(\texttt{twice(twice)}\big)(f).$$

By definition of `twice` and applying this to f:

$$\texttt{twice(twice)(f)} = \texttt{twice}(\texttt{twice}(f)).$$

Now use the exponent view:

$$\texttt{twice}(f) = f^2 \quad \Rightarrow \quad \texttt{twice}(\texttt{twice}(f)) = \texttt{twice}(f^2) = (f^2)^2 = f^4.$$

So **`twice(twice)(f)` is** f^4. If `f(x) = x + 1`, then `twice(twice)(f)(0) = 4`.

Example 4: `thrice(thrice)(f)` First, add parentheses to make the structure clear:

$$\texttt{thrice(thrice)(f)} = \big(\texttt{thrice(thrice)}\big)(f).$$

By definition, `thrice` takes a function h and returns a new function that applies h three times:

$$\texttt{thrice}(h) = h \circ h \circ h = h^3.$$

So:

$$\texttt{thrice(thrice)}(f) = \texttt{thrice}\big(\,\texttt{thrice}(\,\texttt{thrice}(f)\,)\big).$$

Now track powers of f step by step:

$$\begin{aligned} \texttt{thrice}(f) &= f^3, \\ \texttt{thrice}\big(\texttt{thrice}(f)\big) &= \texttt{thrice}(f^3) = (f^3)^3 = f^9, \\ \texttt{thrice}\big(\texttt{thrice}(\texttt{thrice}(f))\big) &= \texttt{thrice}(f^9) = (f^9)^3 = f^{27}. \end{aligned}$$

In particular, if `f(x) = x + 1`, then

$$\texttt{thrice(thrice)(f)(0)} = 27.$$

Example 5: `twice(thrice)(twice)(f)` We start by adding parentheses:

$$\texttt{twice(thrice)(twice)(f)} = \big(\texttt{twice(thrice)}\big)(\texttt{twice})(f).$$

From earlier, we know that for any function g,

$$\texttt{twice(thrice)}(g) = \texttt{thrice}(\texttt{thrice}(g)).$$

So with $g = \texttt{twice}$ we get

$$\big(\texttt{twice(thrice)}\big)(\texttt{twice}) = \texttt{thrice}(\texttt{thrice}(\texttt{twice})).$$

Thus

$$\texttt{twice(thrice)(twice)(f)} = \texttt{thrice}\big(\texttt{thrice}(\texttt{twice})\big)(f).$$

To count how many times f is applied, we analyse the functions:

1. First, recall that:

   - `twice(g)` means "apply g twice", so if $g = f^k$ then `twice(g)` is f^{2k}.
   - `thrice(twice)(g)` is `twice(twice(twice(g)))`.

   Start with a function g that is f^k:

$$\begin{aligned} g &= f^k, \\ \texttt{twice}(g) &\Rightarrow f^{2k}, \\ \texttt{twice}(\texttt{twice}(g)) &\Rightarrow f^{4k}, \\ \texttt{twice}(\texttt{twice}(\texttt{twice}(g))) &\Rightarrow f^{8k}. \end{aligned}$$

   So **`thrice(twice)` multiplies the exponent by 8**: it sends f^k to f^{8k}.

   In particular, for $k = 1$,

$$\texttt{thrice(twice)}(f) = f^8.$$

2. Now look at `thrice(thrice(twice))`. Denote

$$H = \texttt{thrice(twice)}.$$

   We just saw that H multiplies the exponent by 8:

$$H(f^k) = f^{8k}.$$

   Then

$$\texttt{thrice(thrice(twice))} = \texttt{thrice}(H),$$

   which means:

$$\texttt{thrice}(H)(g) = H\big(H(H(g))\big).$$

Apply this to the base function $f = f^1$:

$$\begin{aligned} f^1 &\xrightarrow{H} f^8 \\ &\xrightarrow{H} f^{8\cdot 8} = f^{64} \\ &\xrightarrow{H} f^{64\cdot 8} = f^{512}. \end{aligned}$$

So

$$\texttt{thrice(thrice(twice))}(f) = f^{512}.$$

i.e. the resulting function applies f **512 times** to its input.

If $f(x) = x + 1$, then

$$\texttt{twice(thrice)(twice)(f)}(0) = 512.$$

Quick Summary

In summary,

- `twice` acts as "multiply the exponent of f by 2",
- `thrice` acts as "multiply the exponent of f by 3",
- expressions like `twice(twice)`, `thrice(thrice)`, `twice(thrice)`, etc. are just more complicated ways of chaining these multipliers.

When in doubt:

1. fully parenthesise the expression,
2. compute how many times `f` is actually applied.

Aside: λ-Calculus and Python `lambda`

The word *lambda* in Python's `lambda x: ...` comes from the λ**-calculus**, a tiny mathematical model of computation introduced by Alonzo Church in the 1930s.
Very roughly, λ-calculus has only three ingredients:

- **Variables**: $x, y, z, \ldots$
- **Abstraction**: $\lambda x.\, E$ (a function of one argument x with body E)
- **Application**: $(E_1\ E_2)$ (apply function E_1 to argument E_2)

All programming constructs (numbers, booleans, pairs, recursion, ...) can be encoded using just functions in this system. This is why we say *"functions alone are powerful enough to express any computation"*.

- Our higher-order functions (`map`, `fold`, `compose`, etc.) are direct descendants of ideas from λ-calculus.
- Python's `lambda x: x * x` is inspired by the notation $\lambda x.\, x^2$, but is less general (Python has types, side effects, many built-ins, etc.).

You do *not* need to know formal λ-calculus for this module, but it is the theoretical foundation behind functional programming style we use here.

Additional Reading: SKI Combinator Calculus, see CS1101S 2025 Sem 1 Midterms.

6.4.4 Two-Argument Composition

Sometimes we compose a two-argument function with a one-argument function:

```python
def two_args_compose(f, g):
    return lambda x, y: f(g(x, y))
```

Here g takes two arguments, and f processes the computed result.

```python
def add(x, y): return x + y
def square(z): return z * z

h = two_args_compose(square, add)   # h(x,y) = square(add(x,y))
print(h(1, 2))  # (1 + 2)^2 = 9
```

Arity Checks

When higher-order functions go wrong, the mistake is often that a 2-argument function is treated as if it took 1 argument, or vice versa. For each expression, keep track of:

- "Is this a number?"
- "Is this a 1-argument function?"
- "Is this a 2-argument function?"

This a very common source of runtime errors, as Python does not check for these before allowing the program to run.

In essence, higher-order functions are one of the main tools for writing concise, modular, and reusable programs. You are strongly encouraged to play around with higher-order functions in IDLE to get a feel for how function composition works, as these are very common and tricky questions in examinations.

Looking Forward: Functional Programming

Functional Programming is another popular programming paradigm, as opposed to Object-Oriented Programming. You have gotten your first taste of Functional Programming here, where we chain functions to obtain the results we want.

In CS2030/S, you will formally learn more about Functional Programming as a programming paradigm.