CS1010X: Programming Methodology I
Chapter 2: Functional Abstraction

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 2: Functional Abstraction

Learning Objectives

By the end of this chapter, you should be able to:

- **Understand Abstraction:** View functions as "black boxes" that manage complexity by separating *what* something does from *how* it does it.
- **Define Functions:** Use the `def` keyword to create functions, define formal parameters, and return values.
- **Compose Functions:** Build complex logic by layering simple functions (e.g. `sum_of_squares`).
- **Distinguish Return vs. Print:** Understand the difference between returning a value to the program and displaying text to the user.
- **Master Scope and Environments:** Apply the LEGB rule to understand variable visibility, binding, and shadowing.
- **Utilize Block Structure:** Use nested functions to encapsulate helper logic and prevent namespace pollution.

2.1 Managing Complexity

As programs get larger, they become harder to understand. The primary tool we use to manage this complexity is **functional abstraction**.

The Black Box Concept

A function is like a **black box**.

- **User's Perspective:** We only need to know *what* the box does (inputs and outputs). We do not need to care about the internal wiring.
- **Creator's Perspective:** We design the internal logic once, and then we can forget about the details and just use the box.

For example, when you use `math.sqrt(x)`, you do not need to know the algorithm used to calculate the square root. You simply trust the abstraction.

Looking Forward: Abstraction in Object-Oriented Programming

At the end of the course (and in CS2030/S), you will encounter a programming paradigm known as *objected-oriented programming*. Abstraction is one of the key pillars of object-oriented programming,

2.2 Function Definition

In Python, we create abstractions using the `def` keyword.

2.2.1 Syntax

```
def <name>(<formal parameters>):
    <body>
```

- **Name:** The symbol to be associated with the function (e.g. `square`).
- **Formal Parameters:** Names used within the body to refer to the arguments passed in.
- **Body:** The indented block of code that executes when the function is called.

2.2.2 Example: The Square Function

```
def square(x):
    return x * x
```

We can now use this abstraction to perform compound operations:

```
print(square(5))           # 25
print(square(2+5))         # (5 + 2) * (5 + 2) = 49
print(square(square(3)))   # square(9) -> 81
```

2.2.3 Return VS Print

A function can contain a return statement, as shown above, or simply print a value. There is a difference between `return` and `print`:

- **`return`:** Passes a value back to the caller. This value can be stored in a variable or be used in other expressions.
- **`print`:** Displays characters to the console (side effect). It returns a `None`.

```
def explicit_return(x):
    return x + 1
def just_print(x):
    print(x + 1)
val1 = explicit_return(10) # val1 is now 11
val2 = just_print(10)      # Prints "11", but val2 captures the return
    value
print(val1)       # 11
print(val2)       # None
print(type(val2)) # <class 'NoneType'>
```

The NoneType Trap

If you try to do math on the result of a print function (e.g., `just_print(5)+ 5`), you will get a `TypeError` because you are trying to add an integer to `None`.

2.3 Building Larger Abstractions

We can use existing functions to build larger, more complex functions. This is the essence of modular programming.

Consider calculating the sum of squares $x^2 + y^2$:

```
def sum_of_squares(x, y):
    return square(x) + square(y)

print(sum_of_squares(3, 4)) # 3 ** 2 + 4 ** 2 = 25
```

Here, `sum_of_squares` relies on `square`. If we later improve how `square` works, `sum_of_squares` benefits automatically.

Looking Forward: Multiple Representations

In a later part of this course, we will look at multiple representations of an object (such as rectangular and polar forms for complex numbers). The principle of abstraction will be important so that even if the underlying representation of an object changes, your code does not break.

2.4 Block Structure and Nested Functions

We can define functions inside other functions. This allows us to hide "helper" functions that are irrelevant to the rest of the program, keeping the global namespace clean. This also prevents users from using functions that they are not intended to be able to use.

```
def hypotenuse(a, b):
    def square(x):
        return x * x

    # 'square' is only visible inside 'hypotenuse'
    return (square(a) + square(b)) ** 0.5
```

To the user of `hypotenuse`, the existence of `square` is an irrelevant detail, and we do not want the user to make use of the `square()` function.

Variable Naming

In general, try to use meaningful variable names and avoid unnecessary *overloading* (reusing of variable names). While Python allows you to *shadow* variables (reuse a name in an inner scope), it can make code very hard to read and trace.

2.5 Scope and Environments

When we define variables inside or outside functions, we would like to know where they are visible. The range in which a variable or function can be accessed is known as its **scope**. This is determined by **scoping rules**.

2.5.1 Binding and Scoping

- **Scope:** The region of code where a name is visible.
- **Binding:** The association of a name to a value.
- **Local Scope:** Parameters and variables defined inside a function body are *local* to that function. They are not visible outside of the function.
- **Free variable:** Variable that is referenced in a function but is not defined in that function.

For example,

```
x = 10
y = 1
def square(x):     # This x is "bound" to square and has a local scope.
    print(y)       # This y is a "free variable", it prints 1.
    return x * x   # This x is the "bound" x value in square.
print(square(5))   # Prints 25 (5 * 5).
print(square(x))   # This x refers to the x outside the function. It
    prints 100 (10 * 10).
```

2.6 The LEGB Rule (Not Strictly in Syllabus, but Very Useful)

Python resolves scoping by choosing the *most specific* scope. Specifically, it checks it in this specific order:

1. **L**ocal: Names assigned in any way within a function or class (we will see classes when we cover *object-oriented programming*).
2. **E**nclosing: Names found in the local scope of *enclosing* functions (if this function is nested within another function, as seen above in the case of local functions).
3. **G**lobal: Names assigned at the top-level of a file, or declared global in a function.
4. **B**uilt-in: Special names that Python reserves for itself (e.g. `open`, `range`, `SyntaxError`).

2.6.1 Local and Global Scopes

Let us first ignore the enclosing (E) and built-in (B) scopes in the LEGB rule and only take a look at LG, the local and global scopes.

```
outside = 'outside'
def func():
    print(outside, '[inside func()]')

func()
print(outside, '[outside func()]')
```

What does this print? It prints

```
outside [inside func()]
outside [outside func()]
```

Explanation: We call `func()` first, which is supposed to print the value of `outside`. According to the LEGB rule, the function will first determine if `outside` is defined in its local scope (L). Since `func()` does not define its own `outside` variable, it will look one level up in the global scope (G) in which `outside` has been defined previously.

Now, if we define the variable `outside` in `func()`, what happens?

```
outside = 'outside'
def func():
    outside = 'inside'
    print(outside, '[inside func()]')

func()
print(outside, '[outside func()]')
```

What does this print? It prints

```
inside [inside func()]
outside [outside func()]
```

Explanation: We call `func()` first, which is supposed to print the value of `outside`. According to the LEGB rule, the function will first determine if `outside` is defined in its local scope (L), which it is. So, it will use the definition in its local scope, and print "inside". Note that this does not affect the global variable, which is in a different scope (global scope).

2.6.2 The Global Keyword

Note that is is possible to modify the global variable by re-assigning a new value to it if we use the `global` keyword.

```
outside = 'outside'
def func():
    global outside
    outside = 'inside'
    print(outside, '[inside func()]')

print(outside, '[outside func()]')
func()
print(outside, '[outside func()]')
```

What does this print? It prints

```
outside [outside func()]
inside [inside func()]
inside [outside func()]
```

Explanation: The first printing of `outside` uses the declared global variable. When `func()` is called, it treats the variable `outside` as a global variable due to the `global` keyword. It then assigns "inside" to the global variable `outside`, leading to the next 2 print calls to print "inside".

2.6.3 UnboundLocalError

But we have to be very careful with scoping. It is easy to raise an `UnboundLocalError` if we do not explicitly tell Python that we want to use the global scope and modify a variable's value.

```
outside = 1
def func():
    outside = outside + 1
    print(outside, '[inside func()]')

func()

'''
Traceback (most recent call last):
  File "/tmp/exec_project_6359724/code/main.py", line 6, in <module>
    func()
  File "/tmp/exec_project_6359724/code/main.py", line 3, in func
    outside = outside + 1
              ^^^^^^^
UnboundLocalError: cannot access local variable 'outside' where it is
    not associated with a value
'''
```

Explanation: Note that because we declared a variable `outside` in the local scope of `func()`, on the left hand side of the assignment operator, it is now treated as a local variable. Then,

when the right hand side of the assignment operator is executed, it cannot find any value for `outside` in the local scope, and throws an error!

On the other hand,

```
outside = 1
def func():
    outside = 0
    outside = outside + 1
    print(outside, '[inside func()]')

def func2():
    outside + 1
    print(outside, '[inside func()]')

func() # Prints 1, as expected
func2() # Prints 1, as expected too
```

Note that in the above examples, we have either declared the local variable and assigned a value to it, or have not declared any local variables with the same name. Thus, they execute as expected.

2.6.4 Local, Enclosing and Global Scope

Now, let us consider the enclosing scope.

```
variable = "outside"
def func():
    variable = "enclosed"
    def inner():
        variable = "inner"
        print(variable)
    inner()

func()
```

What does the above function print? It prints "inner".

Explanation: We called `func()`, which defined the variable `variable` locally, then it called the `inner()` function, which defined a variable with the same name too. The print function inside `inner()` found the variable `variable` in its local scope, and therefore printed the value assigned in the inner scope.

```
variable = "outside"
def func():
    variable = "enclosed"
    def inner():
        print(variable)
    inner()

func()
```

The function above prints "enclosed", as expected. The `inner()` function cannot find the variable `variable` defined within its inner scope, so it goes to the enclosing scope, and finds the definition there. Thus, it uses that definition of `variable`.

2.6.5 The Nonlocal Keyword

Similar to the concept of the **`global`** keyword above, we can use the keyword `nonlocal` inside the inner function to explicitly access a variable from the outer (enclosed) scope in order to modify its value.

```
variable = "outside"
def func():
    variable = "enclosed"
    def inner():
        nonlocal variable
        variable = "inner"
        print(variable, "inner()")
    print(variable, "func() first")
    inner()
    print(variable, "func() second")

func()
```

What does it print? It prints

```
enclosed func() first
inner inner()
inner func() second
```

Explanation: Similarly to the **`global`** keyword, `nonlocal` allows access to the variable in its outer scope. The call to `inner()` modifies the enclosing scope's value of `variable`, and hence the second print in `func()` prints "inner".

2.6.6 Local, Enclosing, Global, Built-in

Let us come into the built-in scope. Let us suppose we define our own **`len()`** function, which has the same name as the one defined by Python.

```
variable = "outside"
def len(string):
    return 0

print(len(variable))
```

What will the above print? It will print 0.

Explanation: There is no problem in reusing the built-in names as functions names, although it is highly not recommended. As we go up the LEGB hierarchy, we notice that the function already finds **`len()`** in the global scope, and therefore uses that implementation.

2.7 Order of Execution

A function's body is not evaluated when the function is defined. It is evaluated only when the function is **called**. This is termed *late binding*.

Example 1: Late Binding (Valid)

```
def func():
    return x  # x is not defined yet

x = 1         # x is defined now
print(func()) # Success! Output: 1
```

This works because when `func()` is finally called, Python looks up `x` in the Global scope, and by that time, `x` exists.

Example 2: Missing Definition (Error)

```
def func():
    return x

func()
x = 1

'''
Traceback (most recent call last):
  File "/tmp/exec_project_1023243077/code/main.py", line 5, in <module
    >
    func()
  File "/tmp/exec_project_1023243077/code/main.py", line 3, in func
    return x
           ^
NameError: name 'x' is not defined
'''
```

The definition of `func` is valid, but the call fails because the dependency (`x`) is not satisfied at the moment of execution.

2.8 Practice Questions

Adapted from: AY2024/25 CS1010X Midterms

```
def foo(a, b):
    def bar(c, d):
        if c >= d:
            x = 2
        return x

    x = 1
    return bar(a, b)

print(foo(2, 1))
print(foo(1, 2))
```

What will the above code print?
Answer: 2, Error
Explanation: If we run `foo(1, 2)`, inside `bar()`, since `c (1) < d (2)`, so the `if` block is skipped. `x` is never assigned. Even though there is a `x = 1` in `foo()`, Python sees `x = 2` inside `bar()` and decides `x` must be local to `bar`. Since the local assignment did not happen, `return x` fails with `UnboundLocalError`.

Adapted from: AY2018/19 Sem 1 CS1010S Midterms

```
x = 7
y = "cs1010s"
z = -5
def f(x, y):
    y = y // 5
    x = x * y
    return x
result = f(f(y, x), x-z)
print(result)
```

What will the above code print?
Answer: cs1010scs1010s
Explanation: This question heavily tests your understanding of scoping.
When the inner `f(y, x)` is called, it calls `f("cs1010s", 7)` based on the global scope variables. Then, the `x` and `y` modified are those from the formal parameters. So, `y = 7 // 5 = 1` and therefore `x = "cs1010s"* 1 = "cs1010s"`.
Then, `f("cs1010s", 7 - (-5))` is called, i.e. `f("cs1010s", 12)`. Then, since `y = 12 // 5 = 2`, then `x = "cs1010s"* 2 = "cs1010scs1010s"`. Thus, that is printed.

2.9 Modules and Imports

Python allows us to organize code into separate files called **modules**. This allows us to reuse code and manage large programs effectively.

2.9.1 Import Styles

- **Importing the Module:** Loads the module ‘X‘. You must access objects using dot notation ‘X.name‘.

```
import math
print(math.sqrt(25))
```

- **Importing Specific Objects:** Loads specific functions ‘a‘, ‘b‘, ‘c‘ directly into your namespace.

```
from math import sqrt, pi
print(sqrt(25))
```

- **Importing Everything (Star Import):** Loads *all* public objects from ‘X‘ into your current namespace.

```
from runes import *
show(rcross_bb)
```

The Danger of Star Imports

Using `from module import *` creates references to all public objects. This can cause **name collisions** (e.g., if you have a function named `show` and the module also has `show`, the import will overwrite yours).

2.10 Case Study: The Picture Language (Runes)

To illustrate the elements of programming, we explore a "Picture Language". This allows us to create complex geometric patterns from simple rules.

2.10.1 Primitives

The simplest entities in the language are images.

Table 2.1: Primitive Runes in the Library

Image	Rune Name	Function	Description
	RCross	`rcross_bb`	The standard test image.
	Sail	`sail_bb`	A triangular "sail" shape filling the right half.
	Corner	`corner_bb`	A small triangle in the top-right corner.
	Nova	`nova_bb`	A geometric "star" or nova shape.
	Heart	`heart_bb`	A heart shape.
	Circle	`circle_bb`	A simple filled circle.
	Ribbon	`ribbon_bb`	A spiral ribbon shape.
	Pentagram	`pentagram_bb`	A five-pointed star.
	Black	`black_bb`	A fully filled black square.
	Blank	`blank_bb`	An empty (white) square.

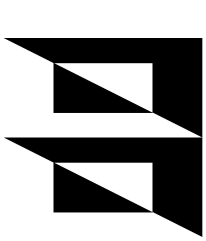
2.10.2 Means of Combination

We combine primitives to create new pictures. The critical property here is the idea of **closure**: The result of a combination is itself a picture, which can be combined further.

In the `runes` library, we have several ways to combine images:

Vertical Combination

- `stack(top, bottom)`: Places `top` above `bottom`. Both images occupy 50% of the vertical space.
- `stack_frac(frac, top, bottom)`: Places `top` above `bottom`, but allows you to specify the split. The `top` image occupies `frac` amount of the height, and `bottom` occupies $1 - \texttt{frac}$.



(a) `stack(rcross_bb, rcross_bb)`

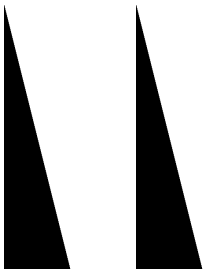
(b) `stack_frac(1/4, rcross_bb, rcross_bb)`

Figure 2.1: Vertical combinations using `rcross_bb`

Horizontal and Depth Combination

While `beside` manages the x-axis, `overlay` functions manage the z-axis (depth).

- `beside(left, right)`: Places `left` next to `right` (50/50 split).
- `overlay(front, back)`: Places `front` on top of `back` (50/50 depth split).
- `overlay_frac(frac, front, back)`: Similar to `stack_frac`, but for depth. The `front` image occupies `frac` amount of the depth, and `back` occupies the rest.



(a) `beside(sail_bb, sail_bb)`

(b) `overlay(pentagram_bb, black_bb)`

(c) `overlay_frac(0.3, pentagram_bb, rcross_bb)`

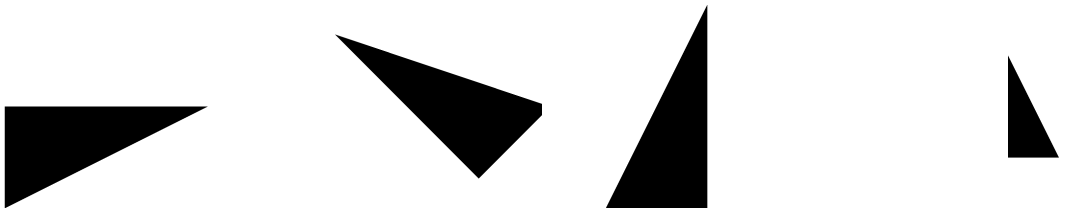
Figure 2.2: Z-axis combinations usually represent depth via shading (darker is closer).

2.10.3 Transformations

These functions accept a single picture and return a transformed version of it.

Table 2.2: Transformation Functions in `runes.py`

Function	Description
`quarter_turn_right(pic)`	Rotates image 90° clockwise.
`quarter_turn_left(pic)`	Rotates image 90° counter-clockwise.
`eighth_turn_left(pic)`	Rotates image 45° counter-clockwise.
`turn_upside_down(pic)`	Rotates image 180°.
`flip_vert(pic)`	Mirrors the image vertically.
`flip_horiz(pic)`	Mirrors the image horizontally.
`scale(ratio, pic)`	Scales image by `ratio` (centered).
`translate(x, y, pic)`	Moves image by `x, y` coordinates.



(a) Rotate Right (b) Eighth Left (c) Flip Horiz (d) Scale (0.5)

Figure 2.3: Various Transformations

2.10.4 Means of Abstraction

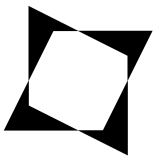
We can now make use of these primitives and means of combinations to manipulate them as units by using abstractions.

Example: Making a Cross

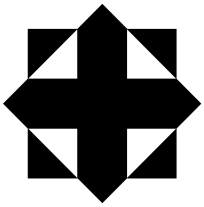
We would like to make a "cross" pattern of images, so instead of manually doing it every time, which is unwieldy, we can instead opt to create a function to do it for us.

```
def make_cross(pic):
    return stack(
        beside(
            quarter_turn_right(pic),
            turn_upside_down(pic)
        ),
        beside(
            pic,
            quarter_turn_left(pic)
        )
    )
```

We can now use `make_cross()` as if it were a primitive. Because the function takes a `pic` argument, it works on *any* picture.



(a) `make_cross(nova_bb)`



(b) `make_cross(rcross_bb)`

Figure 2.4: Abstraction allows us to apply the "cross" pattern to any primitive.

2.11 Introduction to Recursion

What if we want to stack an image n times?

We could write `stack(pic, stack(pic, stack(...)))`, but this is not scalable. We need a way to repeat logic based on a number.

2.11.1 Recursion

Recursion is when a function calls itself. To solve a problem recursively, we break it down into:

1. **Base Case:** The simplest version of the problem (when to stop).
2. **Recursive Step:** Reducing the problem to a smaller version of itself.

2.11.2 Example: `stackn`

We want to stack `pic` n times.

- **Logic:** Stacking n times is the same as placing 1 image on top of a stack of $n-1$ images.
- Of course, simply stacking it will not give you the correct results, since stacking alone splits it into half.

```
def stackn(n, pic):
    if n == 1:
        return pic
    else:
        # stack_frac allows specific sizing.
        # The top takes 1/n of the space.
        # The bottom (recursive call) takes the rest.
        return stack_frac(1/n, pic, stackn(n-1, pic))
```

(a) $n = 1$ (Base Case) (b) $n = 5$ (Recursive)

Figure 2.5: Visualizing `stackn` with `sail_bb`

2.11.3 Generalizing Patterns

We can abstract the pattern of repetition even further. Consider a function that creates a $n \times n$ grid of pictures.

We already have `stackn`, which creates a vertical column of size n. How do we create a grid?

1. Create a **row** of length n.
2. Stack that row n times.

To create a row using only `stackn` (which stacks vertically), we can use a clever trick with rotations:

- We need to rotate the images left first, since the row needs to have the correct orientation after rotating the result of `stackn`.
- We can then stack n rows using `stackn`.

```
def nxn(n, pic):
    # 1. Create a column of rotated images
    col = stackn(n, quarter_turn_left(pic))
    # 2. Rotate the column to become a row
    row = quarter_turn_right(col)
    # 3. Stack n rows to make the grid
    return stackn(n, row)
```

This demonstrates the power of abstraction: we built a complex $n \times n$ geometric structure just by combining simple rotations and our recursive `stackn` abstraction.

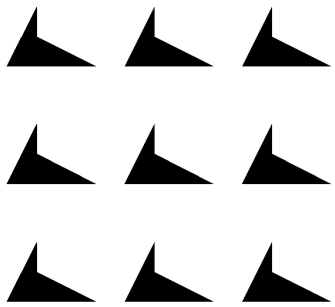


Figure 2.6: `nxn(3, nova_bb)` creates a 3×3 grid.

2.12 Anonymous Functions: Lambda

Sometimes we need a small function for a short period of time, and we do not want to define it formally using `def`. We use **lambda** expressions.

2.12.1 Syntax

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

Note that explicit `return` keyword is not used; the result of the expression is automatically returned.

Table 2.3: Named vs Lambda Functions

Named Function	Lambda Function
`def square(x):` `    return x * x`	`square = lambda x: x * x`

2.12.2 Composing Abstractions

Lambdas allow us to combine existing abstractions into new ones concisely.

For example, we can combine our `nxn` abstraction with the primitive `beside`. By defining a **lambda**, we create a new operation that automatically generates a 2×2 grid and places it side-by-side with itself.

```
# Define a complex operation in one line using lambda
# 1. Create a 2x2 grid
# 2. Place it beside itself
double_grid = lambda p: beside(nxn(2, p), nxn(2, p))
show(double_grid(nova_bb))
```

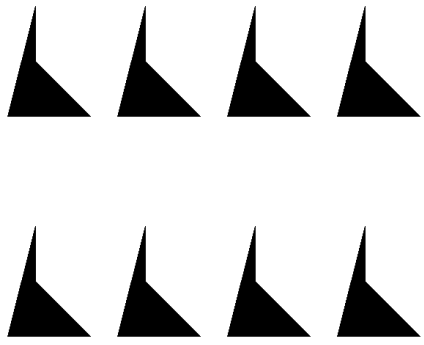


Figure 2.7: The result of the `double_grid` lambda: two 2×2 grids side-by-side.