[1] Building Abstractions with functions

1.1 Getting Started

- Computer fundamentals:
  - representing information
  - specifying logic to process it
  - designing abstractions that manage the complexity of that topic
- Recommended textbook: Structure and Interpretation of Computer Programs (SICP)

1.1.1 Programming in Python

1.1.2 Installing Python 3

1.1.3 Interactive Sessions

- In an interactive Python session, you type some Python code after the ">>>". The Python interpreter reads and executes what you type, carrying out your various commands.
- To start an interactive session, type "python3" at a terminal prompt.
- Interactive controls: each session keeps a history of what you have typed. To access that history, press [Ctrl-p] (previous) or [Ctrl-n] (next). You can exit a session with [Ctrl-d], which discards the history. Up and down arrows also cycle through history on some systems.

1.1.4 First Example

- Statements & Expressions: Python code consists of expressions and statements. Broadly, computer programs consist of instructions to either computer some value or carry out some actions. Statements typically describe actions. Expressions typically describe computations.
- Functions: functions encapsulates logic that manipulates data.
- Objects: an object seamlessly bundles together data and the logic that manipulates that data, in a way that manages the complexity of both.
- Interpreters: evaluating compound expressions requires a precise procedure that interprets code in a predictable way. A program that implements such a procedure, evaluating compound expressions, is called an interpreter. When compared with other computer programs, interpreters for programming languages are unique in their generality.

1.1.5 Errors

- Computers are rigid: even the smallest spelling and formatting changes can cause unexpected output and errors. Learning to interpret errors and diagnose the cause of unexpected errors is called debugging.
- Some guiding principles of debugging are:
  - Test incrementally
  - Isolate errors: trace the error to the smallest fragment of code you can before correcting it
  - Check your assumptions
  - Consult others

1.2 Elements of Programming

- Programs serve to communicate ideas among members for a programming community. Thus, programs must be written for people to read, and only incidentally for machines to execute.
- When we describe a language, we should pay particular attention to the means that the language provide for combining simple ideas to from more complex ideas. Each powerful language has three such mechanisms:
  - primitive expressions and statements: represent the simplest building blocks

means of combination: by which compound elements are built from simpler ones

means of abstraction: by which compound elements can be named and manipulated as units

1.2.1 Expressions

- primitive expressions like numbers in base 10: 42
- compound expressions like numbers combined with mathematical operators: -1 - -1

1.2.2 Call Expressions

- The most important kind of compound expression is a call expression, which applies a function to some arguments.
- A call expression can have subexpressions.
- The order of the arguments in a call function matters.
- Function notation has three principle advantages over the mathematical convention of infix notation

  functions may take an arbitrary number of arguments

  function notation extends in a straightforward way to nested expressions

  mathematical notations has a great variety of forms, while this complexity can be unified via the notation of call expressions

1.2.3 Importing Library Functions

- Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make all of their names available by default. Instead, it organizes the functions and other quantities that it knows into modules., which together comprise the Python library. To use these elements, one imports them. Below are examples:

  ```
  >>> from math import sqrt
  >>> from operator import add, mul
  ```

1.2.4 Names and the Environment

- A critical aspect of a programming language it the means it provides for using names to refer to computational objects. If a value as been given a name, we say that the name binds to the value.
- In Python, we can establish new bindings using the assignment statement, like

  ```
  >>> radius = 10
  >>> radius
  10
  ```

  Here "=" is called the assignment operator. Assignment is our simplest means of abstraction.
- The possibility of binding names to values and later retrieving those values by names means that the interpreter must maintain some sort of memory that keeps track of the names, values and bindings. This memory is called an environment.
- Names can also be bound to functions, like

  ```
  >>> f = max
  >>> f
  <built-in function max>
  >>> f(1, 2, 3, 4)
  4
  ```

- In Python, names are called variable names or variables, because they can be bound to different values in the course of executing a program. When a name is bound to a new value through assignment, it is no longer bound to any previous value. One can even bind built-in names to new values, like max.

- We can assign multiple values to multiple names in a single statement, like
  area, circumference = pi * radius * radius, 2 * pi * radius
  note that after assignment, even if we change the value of radius, area will not change

(because in nature, variables are bound to values, not expressions)

1.2.5 Evaluating Nested Expressions

- To evaluate a call expression, Python will do the following:
  evaluate the operator and operand subexpressions
  apply the function that is the value of the operator subexpression to the arguments that are

the values of the operand subexpressions

- Note that the evaluation procedure is recursive in nature. If we draw each expression that we evaluate, we can visualize the hierarchical structure of this process. This illustrations is called an expression tree. In computer science, trees conventionally grow from the top down. The objects at each point in a tree are called nodes, here they are expressions paired with values.

1.2.6 The Non-Pure Print Function

- Pure functions: have some input (arguments) and return some output (the result of applying them). Pure functions have the property that applying them has no effects beyond returning a value. Moreover, a pure function must always return the same value when passed in same arguments.
- Non-pure function: can generate side effects, which make some changes to the state of the interpreter or computer. A common side effect is to generate additional output beyond the return value, using the print function. The value that print returns is always None (a special Python value that represents nothing)
- Below is a special case:
  >>>print(print(1), print(2))
  1
  2
  None, None

#Note that None evaluates to nothing, and will not be displayed by the interpreter as a value, like
  >>> None

1.3 Defining New Functions

- 3 elements which make Python and other programming languages so powerful:
  Numbers and arithmetic operations are primitive built-in data values and functions
  Nested function application provides a means of combining operations
  Binding names to values provides a limited means of abstraction
- Function definition is a much more powerful abstraction technique by which a name can be bound to compound operation, which can then be referred to as a unit.
- How to define a function:
  def <name> (<formal parameters>):
    return <return expression>

The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied.

- We can use the defined function as a building block in defining other function. Actually, user-defined functions are used in exactly the same way as built-in functions

1.3.1 Environments

- An environment in which an expression is evaluated consists of a sequence of frames, depicted as boxes. Each frame contains bindings, each of which associates a name with its corresponding value.
- global frame
- The name of a function can be repeated twice, once in the frame and again as part of the function itself. The name appearing in the function is called the intrinsic name. The name in a frame is a bound name. There is a difference between the two: different names may refer to the same function, but that function itself has only one intrinsic name. The name bound to a function in a frame is the one used during evaluation. The intrinsic name of a function does not play a role.
- Function signature: a description of the formal parameters of a function, like max(...). Note that the "..." here means max can take an arbitrary number of arguments.

1.3.2 Calling User-Defined Functions

- Applying a user-defined function introduces a second local frame, which is only accessible to that function. To apply a user-defined function to some arguments:

  Bind the arguments to the names of the function's formal parameters in a new local frame.

  Execute the body of the function in the environment that starts with the frame

Note that the first step: bound names, is very tricky! As we have mentioned before, binding is only between values and variable names. So here what we are really doing, is not changing names for the parameters, but evaluating their values and assigning these values to new names! [see HW1 Q5]

- Name evaluation: a name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

1.3.3 Example: Calling a User-Defined Function

1.3.4 Local Names

- One detail of a function's implementation that should not affect the function's behavior is the implementer's choice of names for the function's formal parameters. The simplest function of it is that the parameter names of a function must remain local to the body of the function.

1.3.5 Choosing Names

- Important principles in choosing function and parameter names:

  Function names are lowercase, with words separated by underscores. Descriptive names are encouraged.

  Function names typically evoke operations applied to arguments by the interpreter, or the name of the quantity that results.

  Parameter names are lowercase, with words separated by underscores. Single-word names are preferred.

  Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed.

  Single letter parameter names are acceptable when their role is obvious, but avoid l, O or I to avoid confusion with numerals.

1.3.6 Functions as Abstractions

- To master the use of a functional abstraction, it is often useful to consider 3 core attributes:

  the domain of a function is the set of arguments it can take

  the range of a function is the set of values it can return

the intent of a function is the relationship it computes between inputs and output (as well as any side effects it might generate)

1.3.7 Operators

- distinguish among / (normal division, which results to a floating point, or truediv in operator), // (rounds the result to an integer, or floordiv in operator), and % (mod)

1.4 Designing Functions

- The qualities of good functions:
  each function should have exactly one job
  don't repeat yourself (DRY principle)
  functions should be defined generally

1.4.1 Documentation

- A function definition will often include documentation describing the function, called a docstring, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behavior of the function.
- When you call [help] with the name of a function as an argument, you see its docstring (type [q] to exit), like
  >>> help(pressure)
- Comments in Python can be attached to the end of a line following the "#" symbol

1.4.2 Default Argument Values

- In Python, we can provide default values for the arguments of a function. When calling that function, arguments with default values are optional. If they are not provided, then the default value is bound to the formal parameter name instead. Below is an example:

```
>>> def pressure(v, t, n = 6):
      ……
>>> pressure (1, 273.25) # pass in 2 parameters
2269
>>> pressure (1, 273.15, 6)
2269
```

1.5 Control

- Control statements are statements that control the flow of a program's execution based on the results of logical comparisons.

1.5.1 Statements

- So far, we have seen 3 statements already: def, assignment, and return.
- Rather than being evaluated, statements are executed. Each statement describes some change to the interpreter state, and executing a statement applies that change.

1.5.2 Compound Statements

- In general, Python code is a sequence of statements. A simple statement is a single line that doesn't end in a colon. A compound statement is so-called because it's composed of other statements. Compound statements typically span multiple lines and start with a one-line header ending in colon, which identifies the type of statement. Together, a header and an indented suite of statements is called a clause. A compound statement consists of one or more clauses.

<header>:
    <statement>
    <statement>
    ……
<separating header>:
    <statement>
    <statement>
    …..

- We can catalog the statements we have learnt:
  Expressions, return statement, and assignment statements are simple statements.
  A def statement is a compound statement made up of clauses as the function body.
- We can understand multi-line programs: to execute a sequence of statements, execute the first statement. If that statement does not redirect control, then proceed to execute the rest of the sequence of statements, if any remain.

1.5.3 Defining Functions II: Local Assignment

- Whenever a user-defined function is called, the sequence of clauses in the suite of its definition is executed in a local environment - an environment starting with a local frame created by calling that function. A return statement redirects control: the process of function terminates whenever the first return statement is executed, and the value of the return expression is the returned value of the function being applied.

1.5.4 Conditional Statements

- A conditional statement in Python consists of a series of headers and suits: a required "if" clause, an optional sequence of "elif" clauses, and finally an optional "else" clause:

```
>>> if <expression>:
        <suite>
>>> elif <expression>:
        <suite>
>>> else:
        <suite>
```

- The computational process of executing a conditional clause follows:
  Evaluate th header's expression
  If it is a true value, execute the suite. Then, skip all subsequent clauses in the conditional statement.
- The expressions inside the header statements of conditional blocks are in boolean contexts: their truth value matter to control flow, but otherwise their values are not assigned or returned. Python includes several false values, including 0, None, and the boolean value False. All other numbers are true values.
- Python has two boolean values: True and False.
- Python has three boolean operators: and, or, and not, the first two of which has the so-called short-circuit evaluation behavior. Below are some examples:

```
>>> True and 13
>>> 13 #evaluate each operand; if True, it will return the value of the rightmost operand
>>> 1 and True
>>> True
```

```
>>> 1 and 2 and 3
>>> 3
>>> False or 0
>>> 0 #the left operand is False, so the right operand is evaluated and its value is returned
>>> 0 or False
>>> False
>>> not 10
>>> False # it equals not True
>>> not None
>>> True # it equals not False
```

1.5.5 Iteration

- A while clause structure is like:

```
>>> while <expression>:
        <suite>
```

1.5.6 Testing

- Testing a function is the act of verifying that the function's behavior matches expectations. A test is a mechanism for systematically performing this verification. Tests typically take the form of another function that contains one more sample calls to this function being tested. The returned value is then verified against an expected result. Unlike most functions, which are meant to be general, tests involve selecting and validating calls with specific argument values. Tests also serve as documentation: they demonstrate how to call a function and what argument values are suitable.
- assert: when the expression being asserted evaluates to a true value, executing an assert statement has no effect. When it is a false value, assert causes an error that halts the execution.

```
>>> assert f(8) == 13
>>> def f_test():
        assert f(2) == 1
        assert f(3) == 1
        assert f(50) == 89
```

- Python provides a convenient method for placing simple tests directly in the docstring of a function. THE first line of a docstring should contain a one-line description of the function, followed by a blank line. A detailed description of arguments and behavior may follow. In addition, the docstring may include a sample interactive session that calls the function. Then, the interaction can be verified via the doctest module. Below is an example of usage:

```
>>> def f(n):
        """ Return the sum of the first n elements.

        >>> f(2)
        55
        >>> f(10)
        458
        """
        .....
```

```
>>> from doctest import testmod
>>> testmod()
    TestResults(failed = 0, attempted = 2)
```

- When writing Python in files, rather than directly into the interpreter, tests are typically written in the same file or a neighboring file with the suffix "_test.py". Then all doctests can be run by starting Python with the doctest command line option: [python3 -m doctest (-v) <python_file>]
- A test that applies a single function is called a unit test.
- autograder usage:

1. To prevent the ok autograder from interpreting print statements as output: print with 'DEBUG:' at the front of the outputted line
2. To open n interactive terminal to investigate a failing test for question sum_digits in assignment lab01: [python3 ok -q sum_digits -i]
3. To look at an environment diagram to investigate a failing test for question sum_digits in assignment lab01: [python3 ok -q sum_digits —trace]

1.6 Higher-Order Functions

- We 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.

1.6.1 Functions as Arguments

- Below is an example:

```
>>> def summation(n, term):
        ……
>>> def identity(x):
        ……
>>> def sum_naturals(x, identity):
        ……
```

1.6.2 Functions as General Methods

- 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.
- We learnt about two related big ideas in computer science:

1. Naming and functions allow us to abstract away a vast amount of complexity.
2. 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.

- 1.6.3 Defining Functions III: Nested Definitions
- There are two negative consequences of passing functions as arguments:

1. The global frame becomes cluttered with names of small functions, which must all be unique.
2. We are constrained by particular function signatures

-> Nested function definitions address both of these problems, but require us to enrich our environment model.

- Below is an example:

```
>>> 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)
```

Note that we now place function definitions inside the body of other definitions.

- Like local assignment, local def statements only affect the local frame. These functions are only in scope while sqrt is being evaluated. Locally defined functions also have access to the name bindings in the scope in which they are defined. This discipline of sharing names among nested definitions are called lexical scoping.
- We require two extensions to our environment model to enable lexical scoping:

1. Each user-defined function has a [parent environment where it was defined.
2. When a user-defined function is called, its local frame extends its parent environment.

- Functions 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.

1.6.4 Functions as Returned Values

- Function composition: h(x) = f(g(x))

1.6.5 Example: Newton's Method

1.6.6 Currying

- Given a function f(x, y), we can define a function g such that g(x)(y) = 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. Currying is useful when we require a function that takes in only a single argument (like the map function).
- Inverse Currying transformation:

1. given f, return g <-currying

```
>>> def curry(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

2. given g, return f <-uncurrying

```
>>> def uncurry(g):
    def f(x, y):
        return g(x)(y)
    return f
>>> uncurry(curry(f))
f
```

1.6.7 Lambda Expressions

- 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. Below is an example:

```
>>> def compose(f, g):
```

```
    return lambda x: f(g(x))
```
We can understand the structure of a lambda expression by constructing a corresponding 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, but otherwise it behaves like any other function.

```
>>> s = lambda x: x* x
>>> s
<function <lambda> at 0xf3f490>
>>> s(12)
144
```

1.6.8 Abstractions and First-Class Functions

- Programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to be 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.

1.6.9 Function Decorators

- Python provides special syntax to apply higher-order functions as part of executing a def statement, called a decorator. Below is the most common examples — 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 code, the decorator is equivalent to:

```
>>> def triple(x):
    return 3*x
>>> triple = trace(triple)
```

- The decorator symbol "@" may also be followed by a call expression. The expression following @ is evaluated first (just as the name trace was evaluated first), the def statement second, and finally the result of evaluating the decorator expression is applied to the newly defined function, and the result if bound to the name in the def statement.