1.5 Control

The expressive power of the functions that we can define at this point is very limited, because we have not introduced a way to make comparisons and to perform different operations depending on the result of a comparison. *Control statements* will give us this ability. They are statements that control the flow of a program's execution based on the results of logical comparisons.

Statements differ fundamentally from the expressions that we have studied so far. They have no value. Instead of computing something, executing a control statement determines what the interpreter should do next.

1.5.1 Statements

So far, we have primarily considered how to evaluate expressions. However, we have seen three kinds of statements already: assignment, `def`, and `return` statements. These lines of Python code are not themselves expressions, although they all contain expressions as components.

Rather than being evaluated, statements are *executed*. Each statement describes some change to the interpreter state, and executing a statement applies that change. As we have seen for `return` and assignment statements, executing statements can involve evaluating subexpressions contained within them.

Expressions can also be executed as statements, in which case they are evaluated, but their value is discarded. Executing a pure function has no effect, but executing a non-pure function can cause effects as a consequence of function application.

Consider, for instance,

```
>>> def square(x):
        mul(x, x) # Watch out! This call doesn't return a value.
```

This example is valid Python, but probably not what was intended. The body of the function consists of an expression. An expression by itself is a valid statement, but the effect of the statement is that the `mul`function is called, and the result is discarded. If you want to do something with the result of an expression, you need to say so: you might store it with an assignment statement or return it with a return statement:

```
>>> def square(x):
        return mul(x, x)
```

Sometimes it does make sense to have a function whose body is an expression, when a non-pure function like `print` is called.

```
>>> def print_square(x):
        print(square(x))
```

At its highest level, the Python interpreter's job is to execute programs, composed of statements. However, much of the interesting work of computation comes from evaluating expressions. Statements govern the relationship among different expressions in a program and what happens to their results.

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 is composed of other statements (simple and compound). Compound statements typically span multiple lines and start with a one-line header ending in a 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 understand the statements we have already introduced in these terms.

- Expressions, return statements, and assignment statements are simple statements.
- A `def` statement is a compound statement. The suite that follows the `def` header defines the function body.

Specialized evaluation rules for each kind of header dictate when and if the statements in its suite are executed. We say that the header controls its suite. For example, in the case of `def` statements, we saw that the return expression is not evaluated immediately, but instead stored for later use when the defined function is eventually called.

We can also understand multi-line programs now.

- 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.

This definition exposes the essential structure of a recursively defined *sequence*: a sequence can be decomposed into its first element and the rest of its elements. The "rest" of a sequence of statements is itself a sequence of statements! Thus, we can recursively apply this execution rule. This view of sequences as recursive data structures will appear again in later chapters.

The important consequence of this rule is that statements are executed in order, but later statements may never be reached, because of redirected control.

Practical Guidance. When indenting a suite, all lines must be indented the same amount and in the same way (use spaces, not tabs). Any variation in indentation will cause an error.

1.5.3 Defining Functions II: Local Assignment

Originally, we stated that the body of a user-defined function consisted only of a `return` statement with a single return expression. In fact, functions can define a sequence of operations that extends beyond a single expression.

Whenever a user-defined function is applied, 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 application terminates whenever the first `return`statement is executed, and the value of the `return` expression is the returned value of the function being applied.

Assignment statements can appear within a function body. For instance, this function returns the absolute difference between two quantities as a percentage of the first, using a two-step calculation:

```
1 def percent_difference(x, y):
2     difference = abs(x-y)
3     return 100 * difference / x
4 result = percent_difference(40, 50)
```

Edit code in Online Python Tutor

< Back End Forward >

line that has just executed

next line to execute

Global

percent_difference	
result	25.0

func perc

percent_difference

x	40
y	50
difference	10
Return value	25.0

The effect of an assignment statement is to bind a name to a value in the *first* frame of the current environment. As a consequence, assignment statements within a function body cannot affect the global frame. The fact that functions can only manipulate their local environment is critical to creating *modular*programs, in which pure functions interact only via the values they take and return.

Of course, the `percent_difference` function could be written as a single expression, as shown below, but the return expression is more complex.

```
>>> def percent_difference(x, y):
        return 100 * abs(x-y) / x
>>> percent_difference(40, 50)
25.0
```

So far, local assignment hasn't increased the expressive power of our function definitions. It will do so, when combined with other control statements. In addition, local assignment also plays a critical role in clarifying the meaning of complex expressions by assigning names to intermediate quantities.

1.5.4 Conditional Statements

Video: Show Hide

Python has a built-in function for computing absolute values.

```
>>> abs(-2)
2
```

We would like to be able to implement such a function ourselves, but we have no obvious way to define a function that has a comparison and a choice. We would like to express that if **`x`** is positive, **`abs(x)`** returns **`x`**. Furthermore, if **`x`** is 0, **`abs(x)`** returns 0. Otherwise, **`abs(x)`** returns **`-x`**. In Python, we can express this choice with a conditional statement.

```
1 def absolute_value(x):
2     """Compute abs(x)."""
3     if x > 0:
4         return x
5     elif x == 0:
6         return 0
7     else:
8         return -x
9
10 result = absolute_value(-2)
```

Edit code in Online Python Tutor

< Back End Forward >

Global

absolute_value	
result	2

func absolute_value(x

absolute_value

x	-2
Return value	2

This implementation of `absolute_value` raises several important issues:

Conditional statements. A conditional statement in Python consists of a series of headers and suites: a required `if` clause, an optional sequence of `elif` clauses, and finally an optional `else` clause:

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

When executing a conditional statement, each clause is considered in order. The computational process of executing a conditional clause follows.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite. Then, skip over all subsequent clauses in the conditional statement.

If the `else` clause is reached (which only happens if all `if` and `elif` expressions evaluate to false values), its suite is executed.

Boolean contexts. Above, the execution procedures mention "a false value" and "a true value." The expressions inside the header statements of conditional blocks are said to be in *boolean contexts*: their truth values 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. In Chapter 2, we will see that every built-in kind of data in Python has both true and false values.

Boolean values. Python has two boolean values, called `True` and `False`. Boolean values represent truth values in logical expressions. The built-in comparison operations, `>, <, >=, <=, ==, !=`, return these values.

```
>>> 4 < 2
False
>>> 5 >= 5
True
```

This second example reads "5 is greater than or equal to 5", and corresponds to the function `ge` in the `operator` module.

```
>>> 0 == -0
True
```

This final example reads "0 equals -0", and corresponds to `eq` in the `operator` module. Notice that Python distinguishes assignment (`=`) from equality comparison (`==`), a convention shared across many programming languages.

Boolean operators. Three basic logical operators are also built into Python:

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

Logical expressions have corresponding evaluation procedures. These procedures exploit the fact that the truth value of a logical expression can sometimes be determined without evaluating all of its subexpressions, a feature called *short-circuiting*.

To evaluate the expression `<left> and <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a false value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `<left> or <right>`:

1. Evaluate the subexpression `<left>`.
2. If the result is a true value `v`, then the expression evaluates to `v`.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `not <exp>`:

1. Evaluate `<exp>`; The value is `True` if the result is a false value, and `False` otherwise.

These values, rules, and operators provide us with a way to combine the results of comparisons. Functions that perform comparisons and return boolean values typically begin with `is`, not followed by an underscore (e.g., `isfinite`, `isdigit`, `isinstance`, etc.).

1.5.5 Iteration

Video: Show Hide

In addition to selecting which statements to execute, control statements are used to express repetition. If each line of code we wrote were only executed once, programming would be a very unproductive exercise. Only through repeated execution of statements do we unlock the full potential of computers. We have already seen one form of repetition: a function can be applied many times, although it is only defined once. Iterative control structures are another mechanism for executing the same statements many times.

Consider the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Each value is constructed by repeatedly applying the sum-previous-two rule. The first and second are fixed to 0 and 1. For instance, the eighth Fibonacci number is 13.

We can use a `while` statement to enumerate `n` Fibonacci numbers. We need to track how many values we've created (`k`), along with the kth value (`curr`) and its predecessor (`pred`). Step through this function and observe how the Fibonacci numbers evolve one by one, bound to `curr`.

```
1  def fib(n):
2      """Compute the nth Fibonacci number, for n >= 2."""
3      pred, curr = 0, 1   # Fibonacci numbers 1 and 2
4      k = 2               # Which Fib number is curr?
5      while k < n:
6          pred, curr = curr, pred + curr
7          k = k + 1
8      return curr
9
10 result = fib(8)
```

Global
fib

func fi

Edit code in Online Python Tutor

< Back Step 2 of 25 Forward >

line that has just executed

next line to execute

Remember that commas seperate multiple names and values in an assignment statement. The line:

```
pred, curr = curr, pred + curr
```

has the effect of rebinding the name `pred` to the value of `curr`, and simultaneously rebinding `curr` to the value of `pred + curr`. All of the expressions to the right of `=` are evaluated before any rebinding takes place.

This order of events -- evaluating everything on the right of `=` before updating any bindings on the left -- is essential for correctness of this function.

A `while` clause contains a header expression followed by a suite:

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

To execute a `while` clause:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.

In step 2, the entire suite of the `while` clause is executed before the header expression is evaluated again.

In order to prevent the suite of a `while` clause from being executed indefinitely, the suite should always change some binding in each pass.

A `while` statement that does not terminate is called an infinite loop. Press `<Control>-C` to force Python to stop looping.

1.5.6 Testing

Testing a function is the act of verifying that the function's behavior matches expectations. Our language of functions is now sufficiently complex that we need to start testing our implementations.

A *test* is a mechanism for systematically performing this verification. Tests typically take the form of another function that contains one or more sample calls to the 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 appropriate.

Assertions. Programmers use `assert` statements to verify expectations, such as the output of a function being tested. An `assert` statement has an expression in a boolean context, followed by a quoted line of text (single or double quotes are both fine, but be consistent) that will be displayed if the expression evaluates to a false value.

```
>>> assert fib(8) == 13, 'The 8th Fibonacci number should be 13'
```

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 execution.

A test function for `fib` should test several arguments, including extreme values of `n`.

```
>>> def fib_test():
        assert fib(2) == 1, 'The 2nd Fibonacci number should be 1'
        assert fib(3) == 1, 'The 3rd Fibonacci number should be 1'
        assert fib(50) == 7778742049, 'Error at the 50th Fibonacci number'
```

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`.

Doctests. 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:

```
>>> def sum_naturals(n):
        """Return the sum of the first n natural numbers.

        >>> sum_naturals(10)
        55
        >>> sum_naturals(100)
        5050
        """
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total
```

Then, the interaction can be verified via the doctest module. Below, the `globals` function returns a representation of the global environment, which the interpreter needs in order to evaluate expressions.

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

To verify the doctest interactions for only a single function, we use a `doctest` function called `run_docstring_examples`. This function is (unfortunately) a bit complicated to call. Its first argument is the function to test. The second should always be the result of the expression `globals()`, a built-in function that returns the global environment. The third argument is `True` to indicate that we would like "verbose" output: a catalog of all tests run.

```
>>> from doctest import run_docstring_examples
>>> run_docstring_examples(sum_naturals, globals(), True)
Finding tests in NoName
Trying:
    sum_naturals(10)
Expecting:
    55
ok
Trying:
    sum_naturals(100)
Expecting:
    5050
ok
```

When the return value of a function does not match the expected result, the `run_docstring_examples`function will report this problem as a test failure.

When writing Python in files, all doctests in a file can be run by starting Python with the doctest command line option:

```
python3 -m doctest <python_source_file>
```

The key to effective testing is to write (and run) tests immediately after implementing new functions. It is even good practice to write some tests before you implement, in order to have some example inputs and outputs in your mind. A test that applies a single function is called a *unit test*. Exhaustive unit testing is a hallmark of good program design.

Continue: 1.6 Higher-Order Functions