Introduction to Programming with Python

Qitian Liao

Aug 3, 2020

Contents

1 Functions

1.1 Elements of Programming

Every programming language has three basic mechanisms:
Primitive expressions and statements: This represents the simplest building blocks that the language provides
Means of combination: Compound elements are built from simpler ones
Means of abstraction: Compound elements can be named and manipulated as units.

1.2 Expressions

An expression describes a computation and evaluates to a value.
Primitive Expressions:

1. Number or Numeral. More precisely, the expression that you type consists of the numerals that represent the number in base 10. Expressions representing numbers may be combined with mathematical operators to form a compound expression, which the interpreter will evaluate.

```
>>> 13 + 13
26
```

   These mathematical expressions use infix notation, where the operator (e.g., +, -, *, or /) appears in between the operands (numbers). Python includes many ways to form compound expressions.

2. Name

```
>>> max
<built-in function max>
```

3. String

```
>>> "about"
'about'
```

1.3 Call Expressions

The most important kind of compound expression is a call expression, which applies a function to some arguments.

```
>>> max(6, 8)
8
```

$$\text{Operator}(\text{Operand}, \text{Operand})$$

This call expression has subexpressions: the operator is an expression that precedes parentheses, which enclose a comma-delimited list of operand expressions. The operator specifies a function. When this call expression is evaluated, the function max is called with arguments 6 and 8, and returns a value of 8.

Evaluation procedure for call expressions

1. Evaluate the operator and then the operand subexpressions.

2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpression.

The order of the arguments in a call expression matters. For instance, the function pow raises its first argument to the power of its second argument.

```
>>> pow(2, 5)
32
>>> pow(5, 2)
25
```

Function notation has three principal advantages over the mathematical convention of infix notation.

1. Functions may take an arbitrary number of arguments.

```
>>> max(1, 2, 3, 4)
4
```

2. Function notation extends in a straightforward way to nested expressions, where the elements are themselves compound expressions. In nested call expressions, unlike compound infix expressions, the structure of the nesting is entirely explicit in the parentheses.

```
>>> max(min(1, -2), min(pow(3, 5), -4))
-2
```

3. Mathematical notation has a great variety of forms: multiplication appears between terms, exponents appear as superscripts, division as a horizontal bar, and a square root as a roof with slanted siding. All of this complexity can be unified via the notation of call expressions. While Python supports common mathematical operators using infix notation (like + and -), any operator can be expressed as a function with a name.

1.4 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 about into modules, which together comprise the Python Library. To use these elements, one imports them.

For example, the math module provides a variety of familiar mathematical functions:

```
from math import sqrt
>>> sqrt(25)
5
```

The operator module provides access to functions corresponding to infix operators:

```
from operator import add, sub, mul
>>> add(3, 5)
8
```

An import statement designates a module name (e.g., math), and then lists the named attributes of that module to import (e.g., sqrt). Once a function is imported, it can be called multiple times.

There is no difference between using these operator functions (e.g., add) and the operator symbols themselves (e.g., +). Conventionally, most programmers use symbols and infix notation to express simple arithmetic.

1.5   Exercises

Evaluate the following equations:

```
>>> max(min(3, -2), min(pow(2, 5), -4))
```

2   Names, Assignment and User-Defined Functions

2.1   Names

A critical aspect of a programming language is the means it provides for using names to refer to computational objects. If a value has been given a name, we say that the name binds to the value.

2.2   Assignment

In Python, we can establish new bindings using the assignment statement, which contains a name to the left of = and a value to the right:

```
>>> a = 10
>>> a
10
>>> a + 2
12
```

The = symbol is called the assignment operator in Python. Assignment is our simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations. In this way, complex programs are constructed by building, step by step, computational objects of increasing complexity.

Environment: The possibility of binding names to values and later retrieving those values by name 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. For instance, the name max is bound to the max function we have been using. Functions, unlike numbers, are tricky to render as text, so Python prints an identifying description instead, when asked to describe a function.

```
>>> max
<built-in function max>
```

We can use assignment statements to give new names to existing functions.

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

And successive assignment statements can rebind a name to a new value.

```
>>> f = 2
>>> f
2
```

In Python, names are often 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.

```
>>> max = 5
>>> max
```

```
5
```

After assigning max to 5, the name max is no longer bound to a function, and so attempting to call max(2, 3, 4) will cause an error.

```
>>> max(1, 2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

When executing an assignment statement, Python evaluates the expression to the right of = before changing the binding to the name on the left. Therefore, one can refer to a name in right-side expression, even if it is the name to be bound by the assignment statement.

```
>>> x = 2
>>> x = x + 1
>>> x
3
```

We can also assign multiple values to multiple names in a single statement, where names on the left of = and expressions on the right of = are separated by commas.

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

With multiple assignment, all expressions to the right of = are evaluated before any names to the left are bound to those values. As a result of this rule, swapping the values bound to two names can be performed in a single statement.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

2.3 The Non-pure Print Function

The special **None** represents nothing in Python. A function that does not explicitly return a value will return **None**. **None** is not displayed by the interpreter as the value of an expression.

Pure functions: Functions have some input (their arguments) and return some output (the result of applying them). The built-in function abs can be depicted as a small machine that takes input and produces output. From the output below, we can see that the function **abs** is pure.

```
>>> abs(-1)
1
```

Properties of pure functions:

1. Applying them has no effects beyond returning a value.

2. Must always return the same value when called twice with the same arguments.

Non-pure functions: In addition to returning a value, applying a non-pure function can generate side effects, which make some change 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.

```
>>> print(1, 2, 3)
1 2 3
```

While **print** and **abs** may appear to be similar in these examples, they work in fundamentally different ways. The value that **print** returns is always **None**.The interactive Python interpreter does not automatically print the value **None**. In the case of print, the function itself is printing output as a side effect of being called.
A nested expression of calls to print highlights the non-pure character of the function.

```
>>> print(print(1), print(2))
1
2
None None
```

Let us see another example.

```
>>> two = print(2)
2
>>> print(two)
None
```

2.4   Exercises

For each of the expressions below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. If a function is displayed, write "Function".
Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is **None**.

```
>>> print(1, print(2))
```

```
>>> print(None, print(None))
```

```
>>> print(print(print(2)), print(3))
```

```
print(4, 5) + 1
```

3 Control

3.1 Defining Functions

Function definition is a more powerful means of abstraction, in which it binds names to expressions.

```
def <name>(<formal parameters>):
    return <return expression>
```

Execution procedure for def statements:

1. Create a function with signature <name>(<formal parameters>). Function signature indicates how many arguments a function takes.

2. Set the body of that function to be everything indented after the first line. Function body defines the computation performed when the function is applied.

3. Bind <name> to that function in the current frame.

3.1.1 Environmental Diagrams

1. Every expression is evaluated in the context of an environment. Names have no meaning without an environment.

2. An environment is a sequence of frames, starting with the global frame and followed by a sequence of local frames.

3. A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Look at the following example.

```
>>> from operator import mul
>>> def square(x):
        return mul(x, x)
>>> square(square(3))
```

Sometimes we will have very complicated function calls, and environmental diagrams will come in very handy at that time. Look at the following example. (credit to CS61A Midterm Spring 2015)

```
batman, superman, ivy = 1, -2, -3
def nanana(batman):
    while batman(superman) > ivy:
        def batman(joker):
            return ivy
    return -ivy

def joker(superman):
    if superman(batman):
        ivy = -batman
    return nanana

joker(abs)(abs)
```

We will try to solve this after we cover higher order functions.

3.2 Conditional Statement

A statement is executed by the interpreter to perform an action.

3.2.1 Compound Statements

Above we have seen how we can define functions. And def statements are compound statements.

```
>>> <header>:
>>>     <statement>
>>>     <statement>
>>>     ...
>>> <seperating header>
>>>     <statement>
>>>     <statement>
>>>     ...
```

The first header determines a statement's type. The header of a clause controls the suite that follows. A suite is a sequence of statements. To execute a suite means to execute its sequence of statements, in order.
Execution Rule for a sequence of statements:

1. Execute the first statement.

2. Unless directed otherwise, execute the rest.

3.2.2 Conditional Statements

We start with the following example with 1 statement, 3 clauses, 3 headers, and 3 suites.

```
>>> def abs(x):
...     if x < 0:
...             return -x
...     elif x == 0:
...             return 0
...     else:
...             return x
```

Execution Rule for Conditional Statements:
Each clause is considered in order.

1. Evaluate the header's expression.

2. If it is a true value, execute the suite and skip the remaining clauses.

Syntax Tips:

1. Always starts with "if" clause.

2. Zero or more "elif" clauses.

3. Zero or one "else" clause, always at the end.

3.2.3 Boolean Contexts

False values in Python: False, 0, ", None (more to come)
True values in Python: Anything else.

3.3   Iteration

3.3.1   While Statements

Execution Rule for While Statements:

1. Evaluate the header's expression.

2. If it is a true value, execute the (whole) suite, then return to step 1.