Variables, expressions, and statements

Values and types

A *value* is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and “Hello, World!”

These values belong to different *types*: 2 is an integer, and “Hello, World!” is a *string*, so called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The `print` statement also works for integers. We use the `python` command to start the interpreter.

```
python
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

What about values like “17” and “3.2”? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

They’re strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

Variables

One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value.

An *assignment statement* creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second assigns the integer 17 to `n`; the third assigns the (approximate) value of π to `pi`.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

Variable names and keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, @. But what's wrong with `class`?

It turns out that `class` is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 35 keywords:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Statements

A *statement* is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
```

```
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called *operands*.

The operators `+`, `-`, `*`, `/`, and `**` perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

There has been a change in the division operator between Python 2 and Python 3. In Python 3, the result of this division is a floating point result:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

The division operator in Python 2 would divide two integers and truncate the result to an integer:

```
>>> minute = 59
>>> minute/60
0
```

To obtain the same answer in Python 3 use floored (`//` integer) division.

```
>>> minute = 59
>>> minute//60
0
```

In Python 3 integer division functions much more as you would expect if you entered the expression on a calculator.

Expressions

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

```
>>> 1 + 1
2
```

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 1: Type the following statements in the Python interpreter to see what they do:

```
5
x = 5
x + 1
```

Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

- *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.
- *E*xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.
- *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So the expression `5-3-1` is 1, not 3, because the `5-3` happens first and then `1` is subtracted from 2.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Modulus operator

The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (`%`). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y` .

You can also extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly, `x % 100` yields the last two digits.

String operations

The `+` operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end. For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

The `*` operator also works with strings by multiplying the content of a string by an integer. For example:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `input` that gets input from the keyboard[1]. When this function is called, the program stops and waits for the user to type something. When the user presses `Return` or `Enter`, the program resumes and `input` returns what the user typed as a string.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to `input` to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

The sequence `\n` at the end of the prompt represents a *newline*, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int` using the `int()` function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with
base 10: 'What do you mean, an African or a European swallow?'
```

We will see how to handle this kind of error later.

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and in Python they start with the `#` symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Everything from the `#` to the end of the line is ignored; it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

Choosing mnemonic variable names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
```

```
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[2] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 35 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
    print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
    print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, `pizza` and `slice` are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

```
for word in words:
    print(word)
```

The parts of the code that are defined by Python (`for` , `in` , `print` , and `:`) are in bold and the programmer-chosen variables (`word` and `words`) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like `class` and `yield` , which are keywords, or `odd~job` and `US$` , which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` which is not very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex` .

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate 1/2π, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get π/2, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

Glossary

assignment

A statement that assigns a value to a variable.

concatenate

To join two operands end to end.

comment
Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

evaluate
To simplify an expression by performing the operations in order to yield a single value.

expression
A combination of variables, operators, and values that represents a single result value.

floating point
A type that represents numbers with fractional parts.

integer
A type that represents whole numbers.

keyword
A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

mnemonic
A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator
An operator, denoted with a percent sign (`%`), that works on integers and yields the remainder when one number is divided by another.

operand
One of the values on which an operator operates.

operator
A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence
The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement
A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.

string
A type that represents sequences of characters.

type
A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

value
One of the basic units of data, like a number or string, that a program manipulates.

variable
A name that refers to a value.

Exercises

Exercise 2: Write a program that uses `input` to prompt a user for their name and then welcomes them.

```
Enter your name: Chuck
Hello Chuck
```

Exercise 3: Write a program to prompt the user for hours and rate per hour to compute gross pay.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

We won't worry about making sure our pay has exactly two digits after the decimal place for now. If you want, you can play with the built-in Python `round` function to properly round the resulting pay to two decimal places.

Exercise 4: Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

Use the Python interpreter to check your answers.

Exercise 5: Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.

---

1. In Python 2, this function was named `raw_input`.↩
2. See https://en.wikipedia.org/wiki/Mnemonic for an extended description of the word "mnemonic".↩