

## Chapter 2

# Variables, expressions and statements

### 2.1 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.

```
>>> print 4
4
```

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

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

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

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

## 2.2 Long integers

Python provides a type called long that can handle any size integer. One way to create a long is to write an integer with a capital L at the end:

```
>>> type(1L)
<type 'long'>
```

Any time the result of a computation is too big to be represented with an integer, Python converts the result as a long integer:

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
100000000000L
```

In the first case the result has type int; in the second case it is long.

## 2.3 Variables

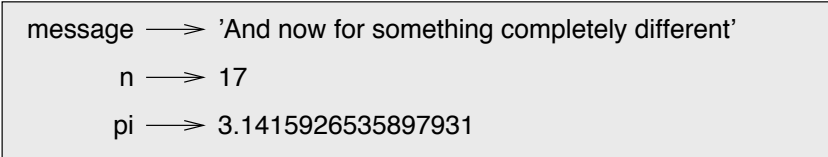
One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

The **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 gives the integer 17 to `n`; the third assigns the (approximate) value of  $\pi$  to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



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

The print statement displays the value of a variable:

```
>>> print n
17
>>> print pi
3.14159265359
```

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

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

## 2.4 Variable names and keywords

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

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

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`.

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 Herpetology'
SyntaxError: invalid syntax
```

76trombones is illegal because it does not begin with a letter. 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 has 31 keywords:

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

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.

## 2.5 Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python 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 itself.

## 2.6 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 following examples demonstrate the arithmetic operators:

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

The symbols +, -, and /, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (\*) is the symbol for multiplication, and \*\* is the symbol for exponentiation.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

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

The value of `minute` is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **floor division**<sup>1</sup>.

When both of the operands are integers, the result is also an integer; floor division chops off the fraction part, so in this example it rounds down to zero.

If either of the operands is a floating-point number, Python performs floating-point division, and the result is a float:

```
>>> minute/60.0
0.98333333333333328
```

The mathematical operators work on long integers; in general any code that works with `int` will also work with `long`.

## 2.7 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:

---

<sup>1</sup>This behavior is likely to change in Python 3.0.

```
>>> 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  $y$ .

Also, you can 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.

## 2.8 Expressions

An **expression** is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

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

Although expressions can contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The following script produces no output at all:

```
17
3.2
'Hello, World!'
1 + 1
```

If you want the script to display the values of these expressions, you have to use `print` statements.

## 2.9 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 the mathematical rules. The acronym **PEMDAS** is a useful way to remember them:

- **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  $(\text{minute} * 100) / 60$ , even though it doesn't change the result.
- **E**xponentiation has the next highest precedence, so  $2**1+1$  is 3 and not 4, and  $3*1**3$  is 3 and not 27.
- **M**ultiplication and **D**ivision (including Modulus) 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 in the expression  $\text{degrees} / 2 * \text{pi}$ , the division happens first and the result is multiplied by pi. If you meant to divide by  $2\pi$ , you should have used parentheses.

## 2.10 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `raw_input` that gets input from the keyboard. 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 `raw_input` returns what the user typed as a string.

```
>>> input = raw_input()
What are you waiting for?
>>> print input
What are you waiting for?
```

Before calling `raw_input`, it is a good idea to print a prompt telling the user what to input. `raw_input` takes a prompt as an argument:

```
>>> name = raw_input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

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`:

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

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

```
>>> speed = raw_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()
```

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

## 2.11 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 they are marked 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 tradeoff.

## 2.12 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` and `SyntaxError: invalid token`, neither of which is very informative.

The run-time 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 `Bob` is not the same as `bob`.

At this point the most likely cause of a semantic error is the order of operations. For example, to evaluate  $\frac{1}{2a}$ , you might be tempted to write

```
>>> 1.0 / 2.0 * a
```

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

And that brings us to the Second Theorem of Debugging:

The only thing worse than getting an error message is not getting an error message.

## 2.13 Glossary

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

**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`).

**integer:** A type that represents whole numbers.

**floating-point:** A type that represents numbers with fractional parts.

**string:** A type that represents sequences of characters.

**variable:** A name that refers to a value.

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

**assignment:** A statement that assigns a value to a variable.

**state diagram:** A graphical representation of a set of variables and the values they refer to.

**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.

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

**operand:** One of the values on which an operator operates.

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

**floor division:** The operation that divides two numbers and chops off the fraction part.

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

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

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

**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.

## 2.14 Exercises

**Exercise 2.1.** Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
delimiter = '.'
```

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. `height/3.0`
3. `width/2.0`
4. `1 + 2 * 5`
5. `delimiter * 5`

**Exercise 2.2.** Practice using the Python interpreter as a calculator:

- If you ran 10 kilometers in 45 minutes 30 seconds, what was your average pace in minutes per mile? What was your average speed in miles per hour? (Hint: there are 1.61 kilometers in a mile).