2. Variables, Expressions, and Statements

Book 1 Think Python, Chapter 2 Variables, expressions and statements

Recall that for simplicity we can consider a variable is a name that refers to a value, the value can be changed and in different data types. In addition, Python is dynamically typed – a variable can be assigned a value without first declaring its data type; Python is weakly typed - a variable can be assigned with values in different data types at different time.

1. Variable Names

1.1 Variables serves two purposes

- Helps the programmer keep track of data that change over time
- Allows the programmer to refer to a complex piece of information with a simple name

1.2 Variable naming rules:

- Name must begin with a letter or _, but can't begin with a number.
- Name can contain any number of letters, digits, or _.
- Names are **case sensitive**, e.g., WEIGHT is different from weight.
- Names cannot be reserved keywords, such as True, False, None, class, if, int, is, for, def, import, and so on.
- Two common naming conventions when a name consists of multiple words:
  [1] Using underscore, _, such as interest_rate and job_rank_code.
  [2] Begin each word with an uppercase, except the first one, such as interestRate and jobRankCode.

```
interest_rate = 0.05
jobRankCode = 'L3'
```

1.3 Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think constants as containers that hold information which cannot be changed later. To distinguish constants from variables, name a constant with upper cases, such as TAX_RATE and THRESHOLD.

```
TAX_RATE = 0.09
THRESHOLD = 60
```

Exercise 1:

In Jupyter Notebook, write statements in test1.ipynb according to the following requirements.

[1] Assign a Boolean value of False to a variable named isSenior.
[2] Assign "GBA" to a variable named courseCode.
[3] Assign 6070 to a variable named courseNumber.
[4] Assign 60 to a constant named PASSING_SCORE.

2. Augmented Assignment

The assignment symbol (**=**) can be combined with the arithmetic and concatenation operators to provide augmented assignment operations. The format is as follows:

<variable> <operator>= <expression>

This statement is equivalent to **<variable> = <variable> <operator> <expression>**

The following statements compare augmented assignments and assignments.

a = 7	a = 7
a += 3	a = a + 3
a −= 3	a = a − 3
a *=3	a = a * 3
a /= 3	a = a / 3
a %= 3	a = a % 3
s = "hi"	s = "hi"
s += " there"	s = s + " there"

3. Expressions

An **expression** is a combination of values, variables, and operators and produces a new value.

An **arithmetic expression** performs arithmetic operations and produces a number.

Each of the following statements is an expression.

```
17
1 + 2
num = 2
num = num + 1
print(num)
```

4. Arithmetic Operators and Their Order

Operator	Meaning	Syntax	Example	Result
+	Addition	a + b	7 + 3	10
−	Subtraction	a − b	7 − 3	4
−	Negation	−a	−7	−7
*	Multiplication	a * b	7 * 3	21
/	Division	a / b	7 / 3	2.33333
//	Quotient (Floor division): divides two numbers and rounds down to an integer	a // b	7 // 3	2
%	Remainder (modulus): divides two numbers and returns the remainder	a % b	7 % 3	1
**	Exponentiation	a ** b	7 ** 3	343

When an expression contains more than one operators, the order of evaluation depends on the following **order of operations**.

[1] Parenthesis
[2] Exponentiation
[3] Unary negation (-)
[4] multiplication, division, and remainder
[5] addition and subtraction

When operands are in different types, the resulting value is of the more general type. If you are not sure, you can use parentheses to make it clear.

```
2 * 3 + 1 # 7
2 * (3 + 1) # 8
1.5 * 2 ** 3 # 12.0
(3 + 1)/-2 # -2.0
-13 % 3 # -1
100 / 5 ** 2 * 2 - 1 # 7.0
```

```
# For multi-line expressions, use a \
1 + 2 * \
3 + 5 # 1 + 2 * 3 + 5 and the result is 12
```

Exercise 2:

```
a = 11
b = 3
```

Given the above statements, do not write any code in Jupyter Notebook and tell the values of the following expressions. Check your answers by running the code in Jupyter Notebook.

-b-a
a/b
a//b
a%b
2**b

Exercise 3:

Given a number in minutes, display the hours and remining minutes. For example, 130 minutes include 2 hours and 10 minutes.

5. String Operations

- Text processing is the most common application of computing.
- In Python, a string literal is a sequence of characters enclosed in either single or double quotation marks. For example,

```
name = 'Sam Jones'
email = "jones@yahoo.com"
```

- '' (two single quotes and nothing in between) and "" (two double quotes) represent an **empty string**. For example,

```
s1 = ''
s2 = ""
```

- Use a pair of ' ' ' (three single quotes) or **"""** (three double quotes) for multi-line paragraphs.

```
s = """This very long sentence extends
all the way
to multiple lines."""
print(s)
```

This very long sentence extends
all the way
to multiple lines

- When a string literal contains double-quotes or single quotation marks:

print("I'm using a single quote in a string!")	A string literal contains a single quote

I'm using a single quote in a string!

print("an 'important' note")	A string literal contains two single quotes

an 'important' note

print('an "important" note')	A string literal contains two double quotes

an "important" note

- The **+** operator performs string **concatenation**

```
first = "Sam"
last = "Jones"
first + last # SamJones and there is no space between Sam and Jones
```

```
"Hi " + "there, " + "Ken!" # Hi there, Ken!
```

- The * operator performs repetition

```
s = "Attention!"
print(s*3) # Attention!Attention!Attention!
```

```
' ' * 10 + 'Python' # '          Python'
```

6. Conversion Functions

In Python, int(), float(), and str() are built-in functions used for type conversion.

int()

- Converts a given value to an integer.
- If the input is a float, it truncates the decimal part (rounds down towards zero).
- If the input is a string representing a whole number, it converts it to an integer.

float():

- Converts a given value to a floating-point number.
- If the input is an integer, it adds a decimal point.
- If the input is a string representing a number (integer or float), it converts it to a float.

str():

- Converts a given value to a string.
- It can convert numbers, booleans, and other data types into their string representations.

Conversion Function	Example	Result
int(<a number or a string>)	int(3.77) int("33") int('2hello')	3 33 ValueError: invalid literal for int() with base 10: '2hello'
float(<a number or a string>)	float(3) float('22') float('2hello')	3.0 22.0 ValueError: could not convert string to float: '2hello'
str(<any value>)	str(3) str(22.0) str('True')	'3' '22.0' 'True'

Note: int() converts a float to an int by truncation, not by rounding. For example, int(6.75) returns 6. If you want to round a number, use the round() function.

round(number, ndigits):

- number: The number to be rounded.
- ndigits (optional): The number of decimal places to round to. If not specified, the number is rounded to the nearest integer.

```
round(6.75) # 7
round(3.14159, 2)  # 3.14
```

In Python variables are bound to specific data types, and type errors will occur if types do not match up as expected in the expression.

```
profit = 1.5
```

```
print('$' + profit)
```

TypeError: can only concatenate str (not "float") to str

To fix the type error, use str() to convert the number to a string, then do string concatenation.

```
print('$' + str(profit))  # $1.5
```

```
"See you at " + 5 + "PM"
```

TypeError: can only concatenate str (not "int") to str

To fix the type error, use str() to convert the number to a string.

```
"See you at " + str(5) + "PM" # See you at 5PM
```

7. Escape Sequences

The purpose of escape sequences is to change output, such as the output format.

Escape Sequence	Meaning
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	The \ character
\’	Single quotation mark
\”	Double quotation mark

```
print("Line 1\nLine 2\t3\t\"4\"")
```

Line 1
Line 2 3 "4"

Exercise 4:

```
product = "Python book"
price = 9.7
quantity = 5
```

Given the above statements, write code in Jupyter Notebook to produce the following output.

Product: Python book
Unit price: $9.7
Quantity: 3
Order total: $29.1

8. Debugging

Three kinds of errors can occur in a program which are syntax errors, runtime errors, and semantic errors.

8.1 Syntax errors

Syntax errors are discovered by the interpreter when it translates the source code into byte code. **Syntax** refers to rules for forming sentences in a language. When Python encounters a syntax error in a program, it halts execution with an error message.

In the following example, the print() statement has a typo.

```
length = 4
print(lenth) # should be print(length)
```

NameError: name 'lenth' is not defined

Attempt to add two numbers, but forgets to include the 2nd one:

```
3 + # should be 3 + 4
```

SyntaxError: invalid syntax

Indentation or indent is significant in Python code.

Each line of code entered must begin in the leftmost column, with no leading spaces. For example, there is an extra space/indent in front of print(n) in the following.

```
n = 2
 print(n)
```

```
  Input In [10]
    print(n)
    ^
IndentationError: unexpected indent
```

The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces.

8.2 Runtime errors

Runtime errors are produced by the Python virtual machine if something goes wrong while the program is running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Most runtime error messages include information about where the error occurred and what functions were executing. For example, an infinite recursion eventually causes the runtime error "maximum recursion depth exceeded".

8.3 Semantic errors

Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. For example, an expression may not be evaluated in the order you expect, yielding an incorrect result.

```
1 + a * b  # a * b is executed first
(1 + a) * b # 1 + a is executed first
```

Debugging tip

When a program generates long error messages which are not useful for debugging, it is very common to use the print() function to print out variable values at some locations in order to trace the program progress and find which part goes wrong.