Fundamentals of Python: First Programs

Chapter 2: Software Development, Data Types, and Expressions

modifications by by

Mr. Dave Clausen
Objectives

After completing this chapter, you will be able to:

• Describe the basic phases of software development: analysis, design, coding, and testing
• Use strings for the terminal input and output of text
• Use integers and floating point numbers in arithmetic operations
• Construct arithmetic expressions
• Initialize and use variables with appropriate names
Objectives (continued)

• Import functions from library modules
• Call functions with arguments and use returned values appropriately
• Construct a simple Python program that performs inputs, calculations, and outputs
• Use docstrings to document Python programs
The Software Development Process

• **Software development**: process of planning and organizing a program
  – Several approaches; one is the **waterfall model**
• Modern software development is usually **incremental** and **iterative**
  – Analysis and design may produce a **prototype** of a system for coding, and then back up to earlier phases to fill in more details after some testing
The Software Development Process (continued)

[FIGURE 2.1] The waterfall model of the software development process
The Software Development Process (continued)

- Programs rarely work as hoped the first time they are run
  - Must perform extensive and careful testing
  - The cost of developing software is not spread equally over the phases
The Software Development Process (continued)

![Figure 2.2] Relative costs of repairing mistakes that are found in different phases
The Software Development Process (continued)

[FIGURE 2.3] Percentage of total cost incurred in each phase of the development process
Case Study: Income Tax Calculator

- Each year nearly everyone faces the unpleasant task of computing his or her income tax return.
- If only it could be done as easily as suggested in this case study.
- We begin with the request:
  - a program that computes a person’s income tax.
Case Study: Analysis

- All taxpayers are charged a flat tax rate of 20%
- Taxpayers are allowed $10,000 standard deduction
- For each dependent, taxpayer is allowed additional $3000 deduction
- Gross income must be entered to nearest penny
- Income tax is expressed as decimal number

```
Enter the gross income: 150000.00
Enter the number of dependents: 3
The income tax is $26200.00
```

[FIGURE 2.4] The user interface for the income tax calculator
Case Study: Design

- Algorithms are often written in a somewhat stylized version of English called **pseudocode**
- Pseudocode for our income tax program:
  - Input the gross income and number of dependents
  - Compute the taxable income using the formula
  - Taxable income = gross income - 10000 - (3000 * number of dependents)
  - Compute the income tax using the formula
  - Tax = taxable income * 0.20
  - Print the tax
Case Study: Implementation (Coding)

""
Program: taxform.py
Author: Ken Lambert

Compute a person's income tax.

1. Significant constants
   tax rate
   standard deduction
   deduction per dependent
2. The inputs are
   gross income
   number of dependents
3. Computations:
   taxable income = gross income - the standard deduction -
                   a deduction for each dependent
   income tax = is a fixed percentage of the taxable income
4. The outputs are
   the income tax
""

# Initialize the constants
TAX_RATE = 0.20
STANDARD_DEDUCTION = 10000.0
DEPENDENT_DEDUCTION = 3000.0

continued
# Request the inputs
grossIncome = float(input("Enter the gross income: "))
numDependents = int(input("Enter the number of dependents: "))

# Compute the income tax
taxableIncome = grossIncome - STANDARD_DEDUCTION - \n    DEPENDENT_DEDUCTION * numDependents
incomeTax = taxableIncome * TAX_RATE

# Display the income tax
print("The income tax is $" + str(incomeTax))

taxform.py
Case Study: Testing

- Even if there are no syntax errors, the program could still have a **logic error** or a **design error**
- May use a **test suite** to test if program is **correct**
### Case Study: Testing (continued)

<table>
<thead>
<tr>
<th>NUMBER OF DEPENDENTS</th>
<th>GROSS INCOME</th>
<th>EXPECTED TAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>-600</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>-1200</td>
</tr>
<tr>
<td>0</td>
<td>20000</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>20000</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>20000</td>
<td>800</td>
</tr>
</tbody>
</table>

*Table 2.1* The test suite for the tax calculator program
Strings, Assignment, and Comments

• Text processing is by far the most common application of computing
  – E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters
Data Types

- A **data type** consists of a set of values and a set of operations that can be performed on those values.
- A **literal** is the way a value of a data type looks to a programmer.
- `int` and `float` are **numeric data types**.
Data Types (continued)

<table>
<thead>
<tr>
<th>TYPE OF DATA</th>
<th>PYTHON TYPE NAME</th>
<th>EXAMPLE LITERALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>int</td>
<td>-1, 0, 1, 2</td>
</tr>
<tr>
<td>Real numbers</td>
<td>float</td>
<td>-0.55, .3333, 3.14, 6.0</td>
</tr>
<tr>
<td>Character strings</td>
<td>str</td>
<td>&quot;Hi&quot;, &quot;,&quot;, 'A', '66'</td>
</tr>
</tbody>
</table>

**[TABLE 2.2]** Literals for some Python data types
String Literals

- In Python, a string literal is a sequence of characters enclosed in single or double quotation marks.
- ' ' and "" represent the **empty string**
- Use ''' and """" for multi-line paragraphs

```python
>>> "I'm using a single quote in this string!"
"I'm using a single quote in this string!"
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!
>>> print('''This very long sentence extends all the way to the next line.''')
This very long sentence extends all the way to the next line.
>>> """"This very long sentence extends all the way to the next line. ""
'This very long sentence extends all the way to\nthe next line.'
>>>```
Escape Sequences

- The newline character `\n` is called an escape sequence

<table>
<thead>
<tr>
<th>ESCAPE SEQUENCE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\</td>
<td>The \ character</td>
</tr>
<tr>
<td>`'</td>
<td>Single quotation mark</td>
</tr>
<tr>
<td>`&quot;</td>
<td>Double quotation mark</td>
</tr>
</tbody>
</table>

[TABLE 2.3] Some escape sequences in Python
String Concatenation

- You can join two or more strings to form a new string using the concatenation operator +
- The * operator allows you to build a string by repeating another string a given number of times.

```python
>>> " " * 10 + "Python"
'Python'
>>> 
```
Variables and the Assignment Statement

• A **variable** associates a name with a value
  – Makes it easy to remember and use later in program

• Variable naming rules:
  – Reserved words cannot be used as variable names
    • Examples: `if`, `def`, and `import`
  – Name must begin with a letter or `_`
  – Name can contain any number of letters, digits, or `_`
  – Names are case sensitive
    • Example: `WEIGHT` is different from `weight`
  – Tip: use “camel casing” (Example: `interestRate`)
Variables and the Assignment Statement (continued)

- Programmers use all uppercase letters for symbolic constants
  - Examples: `TAX_RATE` and `STANDARD_DEDUCTION`
- Variables receive initial values and can be reset to new values with an assignment statement
  \[
  \text{<variable name>} = \text{<expression>}
  \]
  - Subsequent uses of the variable name in expressions are known as variable references

```python
>>> firstName = "Ken"
>>> secondName = "Lambert"
>>> fullName = firstName + " " + secondName
>>> fullName
'Ken Lambert'
>>> 
```
Program Comments and Docstrings

- **Docstring example:**

  ```
  ""
  Program: circle.py
  Author: Ken Lambert
  Last date modified: 2/10/11
  
  The purpose of this program is to compute the area of a circle.
  The input is an integer or floating-point number representing the
  radius of the circle. The output is a floating-point number
  labeled the area of the circle.
  ""
  ```

- **End-of-line comment example:**

  ```
  >>> RATE = 0.85  # Conversion rate for Canadian to US dollars
  ```
Numeric Data Types and Character Sets

• The first applications of computers were to crunch numbers
• The use of numbers in many applications is still very important
Integers

- In real life, the range of integers is infinite
- A computer’s memory places a limit on magnitude of the largest positive and negative integers
  - Python’s `int` typical range: $-2^{31}$ to $2^{31} - 1$
  - $-2,147,483,648$ to $2,147,483,647$
- Integer literals are written without commas
  - $-2147483648$ to $2147483647$
Floating-Point Numbers

• Python uses floating-point numbers to represent real numbers
• Python’s float typical range: $-10^{308}$ to $10^{308}$ and
• Typical precision: 16 digits
### Floating-Point Numbers (continued)

**Table 2.4**: Decimal and scientific notations for floating-point numbers

<table>
<thead>
<tr>
<th>DECIMAL NOTATION</th>
<th>SCIENTIFIC NOTATION</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.78</td>
<td>3.78e0</td>
<td>$3.78 \times 10^0$</td>
</tr>
<tr>
<td>37.8</td>
<td>3.78e1</td>
<td>$3.78 \times 10^1$</td>
</tr>
<tr>
<td>3780.0</td>
<td>3.78e3</td>
<td>$3.78 \times 10^3$</td>
</tr>
<tr>
<td>0.378</td>
<td>3.78e-1</td>
<td>$3.78 \times 10^{-1}$</td>
</tr>
<tr>
<td>0.00378</td>
<td>3.78e-3</td>
<td>$3.78 \times 10^{-3}$</td>
</tr>
</tbody>
</table>
# Character Sets

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
</tr>
<tr>
<td>1</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
</tr>
<tr>
<td>2</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
</tr>
<tr>
<td>3</td>
<td>RS</td>
<td>US</td>
<td>SP</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>4</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
</tr>
<tr>
<td>6</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>‘</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>11</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td>}</td>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

*Table 2.5* The original ASCII character set
Character Sets (continued)

• In Python, character literals look just like string literals and are of the string type
  – They belong to several different character sets, among them the **ASCII set** and the **Unicode set**
• ASCII character set maps to set of integers
• `ord` and `chr` convert characters to and from ASCII

```python
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
```
Expressions

• A literal evaluates to itself
• A variable reference evaluates to the variable’s current value
• **Expressions** provide easy way to perform operations on data values to produce other values
• When entered at Python shell prompt:
  – an expression’s operands are evaluated
  – its operator is then applied to these values to compute the value of the expression
Arithmetic Expressions

- An arithmetic expression consists of operands and operators combined in a manner that is already familiar to you from learning algebra.

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
<th>SYNTAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Negation</td>
<td>-a</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>a ** b</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>a * b</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>a / b</td>
</tr>
<tr>
<td>//</td>
<td>Quotient</td>
<td>a // b</td>
</tr>
<tr>
<td>%</td>
<td>Remainder or modulus</td>
<td>a % b</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>a + b</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>a - b</td>
</tr>
</tbody>
</table>

[Table 2.6] Arithmetic operators
Arithmetic Expressions (continued)

- **Precedence rules:**
  - ** has the highest precedence and is evaluated first
  - Unary negation is evaluated next
  - *, /, and % are evaluated before + and -
  - + and - are evaluated before =
  - With two exceptions, operations of equal precedence are **left associative**, so they are evaluated from left to right
    - exponentiation ** and assignment = are **right associative**
  - You can use () to change the order of evaluation
Arithmetic Expressions (continued)

<table>
<thead>
<tr>
<th>EXPRESSION</th>
<th>EVALUATION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3 * 2</td>
<td>5 + 6</td>
<td>11</td>
</tr>
<tr>
<td>(5 + 3) * 2</td>
<td>8 * 2</td>
<td>16</td>
</tr>
<tr>
<td>6 % 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 * 3 ** 2</td>
<td>2 * 9</td>
<td>18</td>
</tr>
<tr>
<td>-3 ** 2</td>
<td>-(3 ** 2)</td>
<td>-9</td>
</tr>
<tr>
<td>-(3) ** 2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2 ** 3 ** 2</td>
<td>2 ** 9</td>
<td>512</td>
</tr>
<tr>
<td>(2 ** 3) ** 2</td>
<td>8 ** 2</td>
<td>64</td>
</tr>
<tr>
<td>45 / 0</td>
<td>Error: cannot divide by 0</td>
<td></td>
</tr>
<tr>
<td>45 % 0</td>
<td>Error: cannot divide by 0</td>
<td></td>
</tr>
</tbody>
</table>

[Table 2.7] Some arithmetic expressions and their values

45%0 is a **semantic error** – action cannot be carried out

- 45 / 0 is a **semantic error**
Arithmetic Expressions (continued)

• When both operands of an expression are of the same numeric type, the resulting value is also of that type

• When each operand is of a different type, the resulting value is of the more general type
  – Example: `3 // 4` is 0, whereas `3 / 4.0` is 0.75

• For multi-line expressions, use a `\`

```python
>>> 3 + 4 * \n2 ** 5
131
>>> 
```
Mixed-Mode Arithmetic and Type Conversions

- **Mixed-mode arithmetic** involves integers and floating-point numbers:
  ```python
  >>> 3.14 * 3 ** 2
  28.26
  ```

- **Remember**—Python has different operators for quotient and exact division:
  ```python
  3 // 2 * 5.0 yields 1 * 5.0, which yields 5.0
  ```

  ```python
  3 / 2 * 5 yields 1.5 * 5, which yields 7.5
  ```

Tip:
- Use exact division
- Use a **type conversion function** with variables
## Mixed-Mode Arithmetic and Type Conversions (continued)

<table>
<thead>
<tr>
<th>CONVERSION FUNCTION</th>
<th>EXAMPLE USE</th>
<th>VALUE RETURNED</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int(&lt;a number or a string&gt;)</code></td>
<td><code>int(3.77)</code></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><code>int(&quot;33&quot;)</code></td>
<td>33</td>
</tr>
<tr>
<td><code>float(&lt;a number or a string&gt;)</code></td>
<td><code>float(22)</code></td>
<td>22.0</td>
</tr>
<tr>
<td><code>str(&lt;any value&gt;)</code></td>
<td><code>str(99)</code></td>
<td>'99'</td>
</tr>
</tbody>
</table>

**[TABLE 2.8]** Type conversion functions
Mixed-Mode Arithmetic and Type Conversions (continued)

• Note that the \texttt{int} function converts a \texttt{float} to an \texttt{int} by truncation, not by rounding

```python
>>> int(6.75)
6
>>> round(6.75)
7
```
Mixed-Mode Arithmetic and Type Conversions (continued)

- Type conversion also occurs in the construction of strings from numbers and other strings

```python
>>> profit = 1000.55
>>> print('$' + profit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

- Solution: use `str` function

```python
>>> print('$' + str(profit))
$1000.55
```

- Python is a strongly typed programming language
Using Functions and Modules

• Python includes many useful functions, which are organized in libraries of code called modules
• A function is chunk of code that can be called by name to perform a task
Calling Functions: Arguments and Return Values

Functions often require **arguments** or **parameters**
- Arguments may be **optional** or **required**
- When function completes its task, it may **return a value** back to the part of the program that called it

```python
>>> help(round)
Help on built-in function round in module builtin:

round(...) -> floating point number

Round a number to a given precision in decimal digits (default 0 digits).
This returns an int when called with one argument, otherwise the same type as
number. ndigits may be negative.
```
The `math` Module

```python
>>> import math
>>> dir(math)
```

- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource
  - Example: `math.pi`

```python
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```
The `math` Module (continued)

- You can avoid the use of the qualifier with each reference by importing the individual resources

```python
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
```  

- You may import all of a module’s resources to use without the qualifier
  - Example: `from math import *`
  - If you will import all of the functions in module, use:
    ```python
    import math
    ```
The Main Module

• In the case study, earlier in this chapter, we showed how to write documentation for a Python script.
• To differentiate this script from the other modules in a program, we call it the **main module**
  – Like any module, the main module can be imported.

```python
>>> import taxform
Enter the gross income: 120000
Enter the number of dependents: 2
The income tax is $20800.0
```
The Main Module (continued)

```python
>>> help(taxform)

DESCRIPTION
  Program: taxform.py
  Author: Ken

  Compute a person's income tax.

  1. Significant constants
     tax rate
     standard deduction
     deduction per dependent

  2. The inputs are
     gross income
     number of dependents

  3. Computations:
     net income = gross income - the standard deduction -
                   a deduction for each dependent
     income tax = is a fixed percentage of the net income

  4. The outputs are
     the income tax
```
Program Format and Structure

- Start with comment with author’s name, purpose of program, and other relevant information
  - In a docstring
- Then, include statements that:
  - Import any modules needed by program
  - Initialize important variables, suitably commented
  - Prompt the user for input data and save the input data in variables
  - Process the inputs to produce the results
  - Display the results
Running a Script from a Terminal Command Prompt

[FIGURE 2.6] Changing to another directory and listing its contents
Running a Script from a Terminal Command Prompt (continued)

![Figure 2.7](image)

[FIGURE 2.7] Running a Python script in a terminal window
Running a Script from a Terminal Command Prompt (continued)

- Python installations enable you to launch Python scripts by double-clicking the files from the OS’s file browser
  - May require `.py` file type to be set
  - Fly-by-window problem: Window will close automatically
    - Solution: Add an input statement at end of script that pauses until the user presses the enter or return key

```python
input("Please press enter or return to quit the program. ")
```
Summary

- Waterfall model describes software development process in terms of several phases
- Literals are data values that can appear in program
- The string data type is used to represent text for input and output
- Escape characters begin with backslash and represent special characters such as delete key
- A docstring is string enclosed by triple quotation marks and provides program documentation
Summary (continued)

• Comments are pieces of code not evaluated by the interpreter but can be read by programmers to obtain information about a program
• Variables are names that refer to values
• Some data types: int and float
• Arithmetic operators are used to form arithmetic expressions
  – Operators are ranked in precedence
• Mixed-mode operations involve operands of different numeric data types
Summary (continued)

• A function call consists of a function’s name and its arguments or parameters
  – May return a result value to the caller
• Python is a strongly typed language
• A module is a set of resources
  – Can be imported
• A semantic error occurs when the computer cannot perform the requested operation
• A logic error produces incorrect results