A short introduction to Python
1. Introduction

Python is a general-purpose object-oriented, interactive and interpreted programming language with high-level programming capabilities. To learn the Python programming language you do not need any previous programming knowledge or skills. However, the basic understanding of any other programming language will help you to understand the Python programming concepts quickly. In this tutorial, which is used in the course “Computational Thinking”, we will give you a short introduction to Python (version 3) in order to give you a tool to analyze and to design algorithms. So, this tutorial is in no way complete. Please, refer to the official Python documentation at http://docs.python.org if you need more information.
2. **Overview**

Python is designed to be highly readable and is a great language for beginners. It uses English keywords frequently. As mentioned in the introduction, Python is an interpreted programming language. The Python source code is compiled to bytecode as a `.pyc` file, and this bytecode can be interpreted. The Python interpreter can be used in two modes: interactive and script mode.

In the interactive mode, the Python script can be executed directly to the Python prompt without passing the script file to the interpreter. The interactive mode is useful when dealing with small pieces of code as you can type and execute them directly at the Python prompt.

In the script mode, the Python script file is stored with the `.py` extension. The interpreter is then used to execute the contents of the script file. The name of the script file is passed to the interpreter to be executed. The script mode is useful when the code is more than 4 lines and when you want to use the code in future. Example 1 shows how to run the script file `HelloWorld.py` on a UNIX/LINUX system.

*Example*

```
python HelloWorld.py
```

*Example 1: Passing a Python script file to the interpreter*
3. **Reserved keywords**

Like in most programming language, the Python programming language has some reserved keywords which you cannot use as constant, variable or any other identifier names. All the Python reserved keywords contain lowercase letters only. Table 1 shows the Python reserved keywords.

<table>
<thead>
<tr>
<th>as</th>
<th>continue</th>
<th>except</th>
<th>global</th>
<th>lambda</th>
<th>raise</th>
<th>yield</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>def</td>
<td>exec</td>
<td>if</td>
<td>not</td>
<td>return</td>
<td></td>
</tr>
<tr>
<td>assert</td>
<td>del</td>
<td>finally</td>
<td>import</td>
<td>or</td>
<td>try</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>elif</td>
<td>for</td>
<td>in</td>
<td>pass</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>else</td>
<td>from</td>
<td>is</td>
<td>print</td>
<td>with</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1: Reserved keywords in Python*
4. Operators

Operators are special symbols that are used to perform mathematical or logical manipulations. Operands are the values or variables with which the operator is applied to. The values of operands can be manipulated by using the operators.

Example

\[ 10 + 2 = 12 \]

Example 2: Operator and operands

In Example 2 we see two operands (i.e., 10 and 2) and a plus symbol (+) which is the operator. The plus symbol performs the addition. The output of the operation is 12.

The Python programming language supports the following types of operators:

- Arithmetic operators
- Assignment operators
- Comparison (relational) operators
- Logical operators
- Identity operators
- Bitwise operators
- Membership operators

4.1. Arithmetic operators

Table 2 shows the arithmetic operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>10 + 2 = 12</td>
<td>This operator adds the values on either side of the operator.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>6 – 3 = 3</td>
<td>This operator subtracts the right side value from the left side value.</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>8 * 2 = 16</td>
<td>This operator multiplies the values on both sides of the operator.</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>4 / 2 = 2</td>
<td>This operator divides the left side value with the right side value.</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>7 % 2 = 1</td>
<td>This operator returns the remainder by dividing the left side value with right side value.</td>
</tr>
<tr>
<td>**</td>
<td>Exponent</td>
<td>2**3 = 8</td>
<td>This operator calculates the exponential power.</td>
</tr>
</tbody>
</table>
4.2. Assignment operators

Table 3 shows the assignment operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
<td>11 + 2 = 13</td>
<td>This operator calculates the sum of the value of the right side operand and the left side operand.</td>
</tr>
<tr>
<td>+=</td>
<td>Add AND</td>
<td>p += 2</td>
<td>Same as $p = p + 2$</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract AND</td>
<td>p -= 3</td>
<td>Same as $p = p - 3$</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply AND</td>
<td>p *= 4</td>
<td>Same as $p = p * 4$</td>
</tr>
<tr>
<td>/=</td>
<td>Division AND</td>
<td>p /= 5</td>
<td>Same as $p = p /5$</td>
</tr>
<tr>
<td>%=</td>
<td>Modulus AND</td>
<td>p %= 4</td>
<td>Same as $p = p % 4$</td>
</tr>
<tr>
<td>**=</td>
<td>Exponent AND</td>
<td>p **= 3</td>
<td>Same as $p = p ** 3$</td>
</tr>
<tr>
<td>//=</td>
<td>Floor division</td>
<td>p //= 6</td>
<td>Same as $p = p // 6$</td>
</tr>
</tbody>
</table>

Table 3: Assignment operators
4.3. Comparison (relational) operators

Table 4 shows the comparison operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Double equal</td>
<td>$a == b$</td>
<td>TRUE if $a$ is equal to $b$.</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>$a \neq b$</td>
<td>TRUE if $a$ is not equal to $b$.</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
<td>$a &lt;&gt; b$</td>
<td>TRUE if $a$ is not equal to $b$.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>$a &gt; b$</td>
<td>TRUE if $a$ is greater than $b$.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>$a &lt; b$</td>
<td>TRUE if $a$ is less than $b$.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than equal to</td>
<td>$a \leq b$</td>
<td>TRUE if $a$ is less than or equal to $b$.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than equal to</td>
<td>$a \geq b$</td>
<td>TRUE if $a$ is greater than or equal to $b$.</td>
</tr>
</tbody>
</table>

Table 4: Comparison (relational) operators

4.4. Logical operators

Table 5 shows the logical operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>Logical OR</td>
<td>$a \lor b$</td>
<td>TRUE if either $a$ or $b$ is TRUE.</td>
</tr>
<tr>
<td>and</td>
<td>Logical AND</td>
<td>$a \land b$</td>
<td>TRUE if both $a$ and $b$ are TRUE.</td>
</tr>
<tr>
<td>not</td>
<td>Logical NOT</td>
<td>not $a$</td>
<td>TRUE if $a$ is FALSE.</td>
</tr>
</tbody>
</table>

Table 5: Logical operators
4.5. Identity operators
Table 6 shows the identity operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>is</td>
<td>$a$ is $b$</td>
<td>TRUE if values on either side of the operator point to the same object and FALSE otherwise.</td>
</tr>
<tr>
<td>is not</td>
<td>is not</td>
<td>$a$ is not $b$</td>
<td>FALSE if the variables on either side of the operator point to the same object and TRUE otherwise.</td>
</tr>
</tbody>
</table>

Table 6: Identity operators

4.6. Bitwise operators
Table 7 shows the bitwise operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Binary AND</td>
<td>$a$ &amp; $b$</td>
<td>Operator copies a bit to the result if it exists in both operands.</td>
</tr>
<tr>
<td></td>
<td>Binary OR</td>
<td>$a$</td>
<td>Binary OR</td>
</tr>
<tr>
<td></td>
<td>Binary XOR</td>
<td>$a$ ^ $b$</td>
<td>Operator copies the bit if it is set in one operand but not both.</td>
</tr>
<tr>
<td>~</td>
<td>Binary 1s Complement</td>
<td>~$a$</td>
<td>Operator is unary and has the effect of ‘flipping’ bits.</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Binary Left Shift</td>
<td>$a$ &lt;&lt; $b$</td>
<td>The left operands value is moved left by the number of bits specified by the right operand.</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Binary Right Shift</td>
<td>$a$ &gt;&gt; $b$</td>
<td>The left operands value is moved right by the number of bits specified by the right operand.</td>
</tr>
</tbody>
</table>

Table 7: Bitwise operators

4.7. Membership operators
Table 8 shows the membership operators that are supported by Python.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>in</td>
<td>$a$ in $b$</td>
<td>TRUE if it finds a variable in the specified sequence and FALSE otherwise.</td>
</tr>
<tr>
<td>not in</td>
<td>not in</td>
<td>$a$ not in $b$</td>
<td>TRUE if it does not find a variable in the specified sequence and FALSE otherwise.</td>
</tr>
</tbody>
</table>

Table 8: Membership operators
5. Strings

A string is a list of characters represented in either double quote (" ) or single quote (').

Example

```python
str = "Study algorithms."

print (str)  # Prints complete string
print (str[0])  # Prints 1st character of the string
print (str[4:10])  # Prints characters starting from the 5th to 10th
print (str[4:])  # Prints characters starting from the 5th
print (str * 2)  # Prints string two times
print (str + "Hello")  # Prints concatenated string
```

Example 3: Strings

Example 3 shows a few examples of strings. We can take a subset of strings using the slice operator ([ ] and [:]) with index numbers. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. Table 9 shows the output of Example 3.

Output example

```
Study algorithms.
S
y algo
y algorithms.
Study algorithms.Study algorithms.
Study algorithms.Hello
```

Table 9: Output of Example 3
6. Escape characters
The Python programming language has some special characters which are used for special purposes. These characters are represented by a backslash followed by character(s). They can be interpreted using both single and double quote. Table 10 shows the escape characters in Python.

<table>
<thead>
<tr>
<th>Backslash notation</th>
<th>Hexadecimal character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>0x07</td>
<td>Bell or alert</td>
</tr>
<tr>
<td>\b</td>
<td>0x08</td>
<td>Backspace</td>
</tr>
<tr>
<td>\cx</td>
<td></td>
<td>Control-x</td>
</tr>
<tr>
<td>\C-x</td>
<td></td>
<td>Control-x</td>
</tr>
<tr>
<td>\e</td>
<td>0x1b</td>
<td>Escape</td>
</tr>
<tr>
<td>\f</td>
<td>0x0c</td>
<td>Form feed</td>
</tr>
<tr>
<td>\M-\C-x</td>
<td></td>
<td>Meta-Control-x</td>
</tr>
<tr>
<td>\n</td>
<td>0x0a</td>
<td>Newline</td>
</tr>
<tr>
<td>\nnn</td>
<td></td>
<td>Octal notation, n is in the range 0-7</td>
</tr>
<tr>
<td>\r</td>
<td>0x0d</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\s</td>
<td>0x20</td>
<td>Space</td>
</tr>
<tr>
<td>\t</td>
<td>0x09</td>
<td>Tab</td>
</tr>
<tr>
<td>\v</td>
<td>0x0b</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\x</td>
<td></td>
<td>Character x</td>
</tr>
<tr>
<td>\xnn</td>
<td></td>
<td>Hexadecimal notation, n is in the range 0-9, a-f, or A-F</td>
</tr>
</tbody>
</table>

Table 10: Escape characters in Python
7. Numbers

Python supports integers, floating point numbers and complex numbers. These are number-based data types that store various types of numeric values. Number objects are created when we assign a value to them, see Example 4.

Example

var1 = 20
var2 = 7.3

Example 4: Creating number objects

The reference to a number object can be deleted by using the `del` statement, see Example 5.

Example

del var1
del var1, var2

Example 5: Syntax of the `del` statement

7.1. Numerical data types

Table 11 shows the four numerical data types that are supported by Python.

<table>
<thead>
<tr>
<th>Numerical data types</th>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex (complex numbers)</td>
<td>7.15j, 97.1j, 4.78e-8j</td>
<td>Complex numbers are of the form <code>a + bj</code>, where <code>a</code> and <code>b</code> are floating point numbers and <code>J</code> (or <code>j</code>) represents the square root of -1 (imaginary number).</td>
</tr>
<tr>
<td>float (floating point real values)</td>
<td>16.13, -34.78, 2.5e100</td>
<td>Floating points represent real numbers. They are written with a decimal point dividing the integer and fractional parts. Floats may also be written in scientific notation. The <code>E</code> or <code>e</code> indicates the power of 10 (e.g., <code>3.4e2 = 3.4 x 10^2 = 340</code>).</td>
</tr>
<tr>
<td>int (signed integers)</td>
<td>10, -89, 0x260</td>
<td>Signed integers are positive or negative whole numbers with no decimal point.</td>
</tr>
<tr>
<td>long (long integers)</td>
<td>62835472L, 0x19453L, -067674545L</td>
<td>Long integers are integers of unlimited size. They are written like integers and followed by an uppercase or lowercase <code>L</code>.</td>
</tr>
</tbody>
</table>
7.2. Number type conversion

Python converts numbers internally in an expression that contains mixed numerical data types to a common type for evaluation. We can force a number from one type to another type in Python. This is sometimes necessary to satisfy the requirements of an operator or function parameter.

<table>
<thead>
<tr>
<th>Number type conversion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(x)</td>
<td>Is used to convert value x to an integer.</td>
</tr>
<tr>
<td>long(x)</td>
<td>Is used to convert value x to a long integer.</td>
</tr>
<tr>
<td>float(x)</td>
<td>Is used to convert value x to a floating point value.</td>
</tr>
<tr>
<td>complex(x)</td>
<td>Is used to convert value x to a complex number with real part x and imaginary part as 0.</td>
</tr>
<tr>
<td>complex(x, y)</td>
<td>Is used to convert values x and y to a complex number with real part x and imaginary part y.</td>
</tr>
</tbody>
</table>

Table 12: Number type conversion
7.3. Mathematical functions

Table 13 shows the mathematical functions in Python. Note that some functions are not accessible directly. We need to import the math module first in order to call the function using the math static object, e.g., math.log(10).

<table>
<thead>
<tr>
<th>Mathematical function</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>The absolute value of (x): the positive distance between (x) and zero.</td>
<td>abs(-30)</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abs(202.19)</td>
<td>202.19</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>The ceiling of (x): the smallest integer not less than (x).</td>
<td>math.ceil(-30.6)</td>
<td>-30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>math.ceil(20.12)</td>
<td>21</td>
</tr>
<tr>
<td>exp(x)</td>
<td>The exponential of (x): (e^x)</td>
<td>math.exp(4)</td>
<td>54.5981500331</td>
</tr>
<tr>
<td>fabs(x)</td>
<td>The absolute value of (x).</td>
<td>math.fabs(-30)</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>math.fabs(202.19)</td>
<td>202.19</td>
</tr>
<tr>
<td>floor(x)</td>
<td>The floor of (x): the largest integer not greater than (x).</td>
<td>math.floor(-30.6)</td>
<td>-31.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>math.floor(20.12)</td>
<td>20.0</td>
</tr>
<tr>
<td>log(x)</td>
<td>The natural logarithm of (x), for (x &gt; 0).</td>
<td>math.log(10)</td>
<td>2.30258509299</td>
</tr>
<tr>
<td>log10(x)</td>
<td>The base-10 logarithm of (x), for (x &gt; 0).</td>
<td>math.log10(10)</td>
<td>1.0</td>
</tr>
<tr>
<td>max(x1, x2, ...)</td>
<td>The largest of its arguments: the value closest to positive infinity.</td>
<td>max(10, 40, 25)</td>
<td>40</td>
</tr>
<tr>
<td>min(x1, x2, ...)</td>
<td>The smallest of its arguments: the value closest to negative infinity.</td>
<td>min(10, 40, 25)</td>
<td>10</td>
</tr>
<tr>
<td>modf(x)</td>
<td>The fractional and integer parts of (x) in a two-item tuple. Both parts have the same sign as (x). The integer part is returned as a float.</td>
<td>math.modf(10.2)</td>
<td>(0.1999999999999993, 10.0)</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>(x) to the power of (y).</td>
<td>math.pow(10, 2)</td>
<td>100.0</td>
</tr>
<tr>
<td>round(x [, n])</td>
<td>(x) rounded to (n) digits from the decimal point.</td>
<td>round(10.23167, 2)</td>
<td>10.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>round(10.23167, 4)</td>
<td>10.2317</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>The square root of (x), for (x &gt; 0).</td>
<td>math.sqrt(100)</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Table 13: Mathematical functions in Python
7.4. **Trigonometric functions**

Table 14 shows the trigonometric functions in Python. Note that the functions are not accessible directly. We need to import the math module first in order to call the function using the math static object, e.g., math.acos(0).

<table>
<thead>
<tr>
<th>Trigonometric functions</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos((x))</td>
<td>This function returns the arc cosine of (x) in radians.</td>
<td>math.acos(0)</td>
<td>1.57079632679</td>
</tr>
<tr>
<td>asin((x))</td>
<td>This function returns the arc sine of (x) in radians.</td>
<td>math.asin(0)</td>
<td>0.0</td>
</tr>
<tr>
<td>atan((x))</td>
<td>This function returns the arc tangent of (x) in radians.</td>
<td>math.atan(0)</td>
<td>0.0</td>
</tr>
<tr>
<td>atan2((y, x))</td>
<td>This function returns (\arctan(y/x)) in radians.</td>
<td>math.atan2(5, 5)</td>
<td>0.785398163397</td>
</tr>
<tr>
<td>cos((x))</td>
<td>This function returns the cosine of (x) in radians.</td>
<td>math.cos(0)</td>
<td>1.0</td>
</tr>
<tr>
<td>degrees((x))</td>
<td>This function converts angle (x) from radians to degrees.</td>
<td>math.degrees(0)</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>math.degrees(math.pi)</td>
<td>180.0</td>
</tr>
<tr>
<td>hypot((x, y))</td>
<td>This function returns the Euclidean norm, (\sqrt{x^2 + y^2}).</td>
<td>math.hypot(0, 3)</td>
<td>3.0</td>
</tr>
<tr>
<td>sin((x))</td>
<td>This function returns the sine of (x) in radians.</td>
<td>math.sin(0)</td>
<td>0.0</td>
</tr>
<tr>
<td>tan((x))</td>
<td>This function returns the tangent of (x) in radians.</td>
<td>math.tan(0)</td>
<td>0.0</td>
</tr>
<tr>
<td>radians((x))</td>
<td>This function converts angle (x) from degrees to radians.</td>
<td>math.radians(0)</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 14: Trigonometric functions
## 7.5. Random number functions

Table 15 shows some random number functions which are commonly used in Python for, e.g., in games, privacy applications, security, testing, and simulation. Note that the functions are not accessible directly. We need to import the random module first in order to call the function using the random static object, e.g., `random.choice([1, 2, 3])`.

<table>
<thead>
<tr>
<th>Random number function</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>choice(seq)</code></td>
<td>This function returns a random item from a list, tuple, or string.</td>
<td><code>random.choice([1, 2, 3])</code>&lt;br&gt;<code>random.choice('Hello')</code></td>
<td>2&lt;br&gt;e</td>
</tr>
<tr>
<td><code>random()</code></td>
<td>This function returns a random float ( r ), such that ( 0 \leq r &lt; 1 )</td>
<td><code>random.random()</code></td>
<td>0.19661692033</td>
</tr>
<tr>
<td><code>randrange([start,] stop [,step])</code></td>
<td>This function returns a randomly selected element from range(start, stop, step)</td>
<td><code>random.randrange(0, 100, 5)</code></td>
<td>50</td>
</tr>
<tr>
<td><code>seed([x])</code></td>
<td>This function sets the integer starting value used in generating random numbers. The method should be called before any other random module function is called. ( x ) is the seed for the next random number. The method takes system time to generate next random number if ( x ) is omitted.</td>
<td><code>random.seed(5)</code>&lt;br&gt;<code>random.random()</code>&lt;br&gt;<code>random.seed(5)</code>&lt;br&gt;<code>random.random()</code>&lt;br&gt;<code>random.seed(5)</code>&lt;br&gt;<code>random.random()</code></td>
<td>0.62290169489&lt;br&gt;0.62290169489&lt;br&gt;0.62290169489</td>
</tr>
<tr>
<td><code>shuffle(lst)</code></td>
<td>This function randomizes the items of a list in place.</td>
<td><code>list = [4, 56, 20, 34]</code>&lt;br&gt;<code>random.shuffle(list)</code>&lt;br&gt;<code>print(list)</code></td>
<td>[56, 4, 34, 20]</td>
</tr>
<tr>
<td><code>uniform(x, y)</code></td>
<td>This function returns a random float ( r ), such that ( x \leq r &lt; y )</td>
<td><code>random.uniform(10, 25)</code></td>
<td>18.6310645777</td>
</tr>
</tbody>
</table>

*Table 15: Random number functions*
8. Data types

Data types are an important concept in almost all programming languages. They represent a type of the data which can be processed in a computer program. Data types tell the interpreter how the programmer intends to use the data. It is important to specify the type of data when we write a computer program to process different types of data such as strings and integers. If we lack to specify the type of data, the computer will not understand how the different operations should be performed on the given data. The Python interpreter can determine which data type we are storing. So, there is no need to specify the data type in Python as it understands a given data type automatically. Many data types are available in Python. Some important ones we list below.

- Numbers (int, float, long, complex)
- Sequences (strings, bytes/byte array, lists, tuples)
- Boolean (true/false)
- Sets
- Dictionaries
- Module
- Function
- Class
- Method
- File
9. Variables

In simplest terms, we could say that a variable is just a box that you can use to put stuff in. To distinguish the different boxes (i.e., variables) we label the boxes. This means every variable has a name which describes what the variable is storing. We can view the content of the box and we can change the content just by calling the box label.

Formally, we can say that variables are reserved memory locations to store values (e.g., a letter or a number). When we create a variable we reserve some space in memory. These variables hold values temporarily during program execution. Based on the data type of a variable, the Python interpreter allocates memory and decides what can be stored in the reserved memory. Different data types like integers, decimals, characters, etc. can be stored in these variables.

Variables do not need to be declared in Python as the Python interpreter determines what type of data is stored. To assign values to a variable we use an equal sign (=) in the Python programming language. The equal sign assigns the value of right side operand to left side operand. The left side operand is the name of the variable and the right side operand is the assigned value.

Example

```python
name = "John"  # A string
height = "200"  # An integer
distance = "4.5"  # A floating point
```

Example 6: Assigning values to variables

To create variable names in Python we should comply with the following rules:
- Variable names must begin with a letter or underscore.
- Variable names are case-sensitive.
- A variable name does not contain spaces.
- A variable name can only contain alphanumerical characters (i.e., a-z, A-Z, and 0-9) and underscore (_).
- Reserved keywords cannot be used as variable names.
10. Lists
The lists data structure in Python is used to organize data in a single set. Lists are sequences, just like tuples. Each element of a sequence is assigned a number, i.e., the index number or the position of the element. The first index number is zero, the second index number is one, the third index number is two, and so forth. On all sequence types (e.g., lists or tuples) we can apply operations like indexing, adding, multiplying, slicing, and checking for membership.

10.1. Creating lists
A list can be created by putting different comma-separated values or elements between square brackets.

Example

```python
list1 = ['John', 'Paul', 'Catherine']
list2 = [100, 300, 500]
list3 = [20, 10, "Tina", "John"]
```

Example 7: Creating a list

10.2. Accessing values in lists
Values in a list can be accessed by square brackets for slicing along with index numbers. For example, $L[i]$ represents the value at index $i$ in list $L$.

Example

```python
list1 = ['John', 'Paul', 'Catherine']
list2 = [100, 300, 500]
list3 = [20, 10, "Tina", "John"]

print("list1[0]=" , list1[0])
print("list2[2]=" , list2[2])
print("list3[1:3]=" , list3[1:3])
```

Example 8: Accessing values in lists

Output example

```
list1[0] = John
list2[2] = 500
list3[1:3] = [10, 'Tina']
```

Table 16: Output of Example 8
10.3. Updating lists

Example 9 shows how single or multiple values can be updated in a list.

Example

```python
list1 = ['John', 'Paul', 'Catherine']
print ("The third value in list1 is:")
print (list1[2])
print ("The value Catherine is updated in list1 with Carin:")
list1[2]= 'Carin'
print (list1)
```

Output example

```
The third value in list1 is:
Catherine
The value Catherine is updated in list1 with Carin:
['John', 'Paul', 'Carin']
```

Table 17: Output of Example 9

10.4. Deleting values from lists

The `del`-statement can be used to remove a value from a list by `del list_name[index]`.

Example

```python
list1 = ['John', 'Paul', 'Catherine']
print ("list1 before deleting values:")
print (list1)
del list1[2]
print ("list1 after deleting value at index 2:")
print (list1)
```
Example 10: Deleting values from lists

Output example

| list1 before deleting values: | ['John', 'Paul', 'Catherine'] |
| list1 after deleting value at index 2: | ['John', 'Paul'] |

Table 18: Output of Example 10

10.5. Basic list operations
The operators * (asterisk) and + (plus sign) work in lists almost the same as in strings. The * stands for repetition and the + stands for concatenation. The result is a new list instead of a string. Table 19 shows the basic list operations in Python.

<table>
<thead>
<tr>
<th>Basic list operations</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>len([x1, x2, x3,...])</td>
<td>Length</td>
<td>len([1, 2, 3, 4])</td>
<td>4</td>
</tr>
<tr>
<td>[x1, x2,..] + [x3, x4,..]</td>
<td>Concatenation</td>
<td>[1, 2] + [3, 4]</td>
<td>[1, 2,3,4]</td>
</tr>
<tr>
<td>[x] * y</td>
<td>Repetition</td>
<td>['hello'] * 3</td>
<td>['hello', 'hello', 'hello']</td>
</tr>
<tr>
<td>y in [x, y,z]</td>
<td>Membership</td>
<td>2 in [1, 2, 3, 4]</td>
<td>True</td>
</tr>
<tr>
<td>for x in [y1, y2,...]: print (x, end=&quot; &quot;)</td>
<td>Iteration</td>
<td>for x in [1, 2, 3]: print (x, end=&quot; &quot;)</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

Table 19: Basic list operations in Python

10.6. Built-in list functions
Python has some built-in list functions which are shown in Table 20.

<table>
<thead>
<tr>
<th>List functions</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(list)</td>
<td>This function returns the number of elements in the list.</td>
<td>list = [1, 2, 3] print (len(list))</td>
<td>3</td>
</tr>
<tr>
<td>max(list)</td>
<td>This function returns the elements from the list with</td>
<td>list = [10, 280, 38] print (max(list))</td>
<td>280</td>
</tr>
</tbody>
</table>
### 10.7. Built-in list methods

Table 21 shows the built-in list methods in Python.

<table>
<thead>
<tr>
<th>List methods</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>list.append(obj)</td>
<td>This method appends an object \textit{obj} to \textit{list}.</td>
<td>list1 = [1, 2]</td>
<td>[1, 2, 3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.append(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
<tr>
<td>list.count(obj)</td>
<td>This method returns count of how many times \textit{obj} occurs in \textit{list}.</td>
<td>list1 = [1, 2, 1, 1]</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1.count(1))</td>
<td></td>
</tr>
<tr>
<td>list.extend(seq)</td>
<td>This method appends the contents of \textit{seq} to \textit{list}.</td>
<td>list1 = [1, 2]</td>
<td>[1, 2, 3, 4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list2 = [3, 4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.extend(list2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
<tr>
<td>list.index(obj)</td>
<td>This method returns the lowest index in \textit{list} that \textit{obj} appears.</td>
<td>list1 = ['Zara', 'John', 'Joe']</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1.index('John'))</td>
<td></td>
</tr>
<tr>
<td>list.insert(index, obj)</td>
<td>This method inserts object \textit{obj} into \textit{list} at offset \textit{index}.</td>
<td>list1 = ['a', 'b', 'd', 'e']</td>
<td>['a', 'b', 'c', 'd', 'e']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.insert(2, 'c')</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
<tr>
<td>list.pop(obj=list[-1])</td>
<td>This method removes and returns last object or \textit{obj} from \textit{list}.</td>
<td>list1 = ['a', 'b', 'c', 'd', 'e']</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1.pop())</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1.pop(2))</td>
<td>d</td>
</tr>
<tr>
<td>list.remove(obj)</td>
<td>This method removes object \textit{obj} from \textit{list}.</td>
<td>list1 = ['a', 'b', 'c', 'd', 'e']</td>
<td>['a', 'b', 'd', 'e']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.remove('c')</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
<tr>
<td>list.reverse()</td>
<td>This method reverses objects of \textit{list} in place.</td>
<td>list1 = ['a', 'b', 'c', 'd', 'e']</td>
<td>['e', 'd', 'c', 'b', 'a']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.reverse()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
<tr>
<td>list.sort([func])</td>
<td>This method sorts objects of \textit{list}; use compare \textit{func} if given.</td>
<td>list1 = [34, 23, 'd', 'a', 'j']</td>
<td>[23, 34, 'a', 'd', 'j']</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list1.sort()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>print (list1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 20: Built-in list functions in Python

Table 21: Built-in list methods in Python
11. Tuples
Just like lists, tuples are sequences of objects which are immutable. Once a tuple is created it cannot be changed. The differences between tuples and lists are:

- tuples cannot be changed unlike lists,
- tuples are enclosed within parentheses instead of square brackets,
- values/elements of the tuples must have a defined order.

Like in lists, in tuples each element of a sequence is assigned a number, i.e., the index number or the position of the element. The first index number is zero, the second index number is one, the third index number is two, and so forth. We can apply operations like indexing, adding, multiplying, slicing, and checking for membership on tuples and other sequence types (e.g., lists).

11.1. Creating tuples
A tuple can be created by putting different comma-separated values or elements between parentheses.

Example

```python
list1 = ('John', 'Paul', 'Catherine')
list2 = (100, 300, 500)
list3 = (20, 10, "Tina", "John")
```

Example 11: Creating a tuple

11.2. Accessing values in tuples
Values in a tuple can be accessed by square brackets for slicing along with index numbers. For example, $T[i]$ represents the value at index $i$ in tuple $T$.

Example

```python
tuple1 = ('John', 'Paul', 'Catherine')
tuple2 = (100, 300, 500)
tuple3 = (20, 10, "Tina", "John")

print ("tuple1[0]=" , tuple1[0])
print ("tuple2[2]=" , tuple2[2])
print ("tuple3[1:3]=" , tuple3[1:3])
```

Example 12: Accessing values in tuples
11.3. Updating tuples
Tuples are immutable which means that we cannot update or change the values of tuple elements. However, we can join tuples to create a new tuple as Example 13 shows.

Example

```python
tuple1 = ('John', 'Paul', 'Catherine')
tuple2 = (23, 67, 78)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Output example

```
('John', 'Paul', 'Catherine', 23, 67, 78)
```

11.4. Deleting values from tuples
We cannot remove individual values from a tuple. But we can remove a whole tuple with the `del`-statement by `del tuple_name`.

Example

```python
tuple1 = ('John', 'Paul', 'Catherine')
delete tuple1
```

Example 14: Deleting a tuple in Python
11.5. Basic tuple operations

The operators * (asterisk) and + (plus sign) work in tuples almost the same as in strings. The * stands for repetition and the + stands for concatenation. The result is a new tuple instead of a string. Table 24 shows the basic tuple operations in Python.

<table>
<thead>
<tr>
<th>Basic tuple operations</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>len((x1, x2, x3,...))</td>
<td>Length</td>
<td>len((1, 2, 3, 4))</td>
<td>4</td>
</tr>
<tr>
<td>(x1, x2,...) + (x3, x4,...)</td>
<td>Concatenation</td>
<td>(1, 2) + (3, 4)</td>
<td>(1, 2,3,4)</td>
</tr>
<tr>
<td>(x) * y</td>
<td>Repetition</td>
<td>('hello') * 3</td>
<td>('hello', 'hello', 'hello')</td>
</tr>
<tr>
<td>y in (x, y,z)</td>
<td>Membership</td>
<td>2 in (1, 2, 3, 4)</td>
<td>True</td>
</tr>
<tr>
<td>for x in (y1, y2,...): print (x, end=&quot; &quot;)</td>
<td>Iteration</td>
<td>for x in (1, 2, 3): print (x, end=&quot; &quot;)</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

Table 24: Basic tuple operations in Python

11.6. Built-in tuple functions

Table 25 shows the built-in tuple functions in Python.

<table>
<thead>
<tr>
<th>Tuple functions</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(tuple)</td>
<td>This function returns the number of elements in the tuple.</td>
<td>tuple = (1, 2, 3) print (len(tuple))</td>
<td>3</td>
</tr>
<tr>
<td>max(tuple)</td>
<td>This function returns the elements from the tuple with maximum value.</td>
<td>tuple = (10, 280, 38) print (max(tuple))</td>
<td>280</td>
</tr>
<tr>
<td>min(tuple)</td>
<td>This function returns the elements from the tuple with minimum value.</td>
<td>tuple = (10, 280, 38) print (min(tuple))</td>
<td>10</td>
</tr>
<tr>
<td>tuple(seq)</td>
<td>This function converts a list into tuple.</td>
<td>list = ['x', 1, 'r', 10] tuple = tuple(list) print (tuple)</td>
<td>('x', 1, 'r', 10)</td>
</tr>
</tbody>
</table>

Table 25: Built-in tuple functions in Python
12. Dictionary
Dictionaries are the fundamental data structure in Python. They have been heavily optimized for memory overhead and lookup speed efficiency. A dictionary consists of keys and values, where each key is unique and maps a value. The combination of a key and its value is called a key-value pair or item. Each key is separated from its value by a colon (:) and the items are separated by a comma. A dictionary is enclosed by curly braces ({}). The keys of a dictionary should be of an immutable data type such as strings, numbers, or tuples. The values can be of any types.

12.1. Creating dictionaries
Example 15 shows a simple way to create a dictionary in Python.

Example
```
dict = {'Name': 'Catherine', 'Age': 18, 'Gender': 'Female'}
```
Example 15: Creating a dictionary

12.2. Accessing values in dictionaries
You can access values in a dictionary by using square brackets along with the key name to obtain the value.

Example
```
dict = {'Name': 'Catherine', 'Age': 18, 'Gender': 'Female'}
print ("Name: ", dict['Name'])
print ("Age: ", dict['Age'])
print ("Gender: ", dict['Gender'])
```
Example 16: Accessing values in a dictionary

Output example
```
Name: Catherine
Age: 18
Gender: Female
```
Table 26: Output of Example 16
12.3. Updating dictionaries
A dictionary can be updated by
- adding a new key-value pair or a new entry,
- modifying an existing entry, or
- deleting an existing entry.

Example 17 shows how you can update an existing entry and how you can add an entry in a dictionary.

Example

```python
dict = {'Name': 'Catherine', 'Age': 18, 'Gender': 'Female'}
dict['Age'] = 20                   # update an existing entry
dict['Residence'] = "Amsterdam"    # Add a new entry

print ("Name: ", dict['Name'])
print ("Age: ", dict['Age'])
print ("Gender: ", dict['Gender'])
print ("Residence: ", dict['Residence'])
```

Example 17: Updating dictionaries

Output example

Name: Catherine
Age: 20
Gender: Female
Residence: Amsterdam

Table 27: Output of Example 17

12.4. Deleting dictionaries
You can either remove individual entries from a dictionary with the del-statement or clear the entire contents of a dictionary with the `dict.clear()` method. The entire dictionary can be deleted with the del-statement by `del dictionary_name`. See also Example 18.

Example

```python
dict = {'Name': 'Catherine', 'Age': 18, 'Gender': 'Female'}
del dict['Age']                   # remove entry with key 'Age'
dict.clear()                      # remove all entries in dict
del dict                         # delete entire dictionary
```

Example 18: Deleting entries from dictionaries
12.5. Built-in dictionary functions

Python has some built-in dictionary functions which are shown in Table 28.

<table>
<thead>
<tr>
<th>Dictionary functions</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
</table>
| len(dict)            | This function returns the number of items in the dictionary. | `dict = {'Name': 'Jean', 'Age': 17}
print (len(dict))` | `2` |
| str(dict)            | This function produces a printable string representation of a dictionary. | `dict = {'Name': 'Jean', 'Age': 17}
print (str(dict))` | `{'Name': 'Jean', 'Age': 17}` |
| type(variable)       | This function returns the type of the passed variable. If the passed variable is dictionary then it would return a dictionary type. | `dict = {'Name': 'Jean', 'Age': 17}
print (type(dict))` | `<type 'dict'>` |

Table 28: Built-in dictionary functions in Python
### 12.6. Built-in dictionary methods
Table 29 shows the built-in dictionary methods in Python.

<table>
<thead>
<tr>
<th>Dictionary methods</th>
<th>Description</th>
<th>Example</th>
<th>Output example</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict.clear()</td>
<td>This method removes all elements of dictionary dict.</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dict.clear()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict)</code></td>
<td></td>
</tr>
<tr>
<td>dict.copy()</td>
<td>This method returns a shallow copy of dictionary dict.</td>
<td><code>dict1 = {'Name': 'Tina', 'Age': 18}</code></td>
<td>`{Name': 'Tina', 'Age': 18, }</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dict2 = dict1.copy()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(str(dict2))</code></td>
<td></td>
</tr>
<tr>
<td>dict.fromkeys(seq[, value])</td>
<td>This method creates a new dictionary with keys from seq and values set to value if provided.</td>
<td><code>seq = ('Name', 'Age')</code></td>
<td>`{Name': 20, 'Age': 20}</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dict = dict.fromkeys(seq, 20)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(str(dict))</code></td>
<td></td>
</tr>
<tr>
<td>dict.get(key, default=none)</td>
<td>This method returns a value for the given key. Default is the value that is returned in case the key does not exist.</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td><code>Tina</code> Does not exist</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.get('Name'))</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.get('Gender', &quot;Does not exist&quot;))</code></td>
<td></td>
</tr>
<tr>
<td>dict.items()</td>
<td>This method returns a view of dict's (key, value) tuple pairs</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td>dict_items(['Name', 'Tina'], ('Age', 18))</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.items())</code></td>
<td></td>
</tr>
<tr>
<td>dict.keys()</td>
<td>This method returns a view of all available keys in the dictionary.</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td>dict_keys(['Name', 'Age'])</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.keys())</code></td>
<td></td>
</tr>
<tr>
<td>dict.setdefault(key, default=none)</td>
<td>This method is similar to get(), but will set dict[key]=default if key is not already in dict.</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td><code>Tina</code> None</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.setdefault('Name', &quot;None&quot;))</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.setdefault('Gender', &quot;None&quot;))</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(&quot;New dict: %s&quot; % dict)</code></td>
<td></td>
</tr>
<tr>
<td>dict.update(dict2)</td>
<td>This method adds dictionary dict2's key-values pairs to dict1.</td>
<td><code>dict1 = {'Name': 'Tina', 'Age': 18}</code></td>
<td>`{Name': 'Tina', 'Age': 18, 'Gender': 'female'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dict2 = {'Gender': 'female'}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dict1.update(dict2)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict1)</code></td>
<td></td>
</tr>
<tr>
<td>dict.values()</td>
<td>This method returns a view of all the values in a given dictionary.</td>
<td><code>dict = {'Name': 'Tina', 'Age': 18}</code></td>
<td>dict_values(['Tina', 18])</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>print(dict.values())</code></td>
<td></td>
</tr>
</tbody>
</table>

*Table 29: Built-in dictionary methods in Python*
13. Decision-making

In real life, we encounter sometimes situations when we need to make decisions. Based on these decisions we decide what to do next. Likewise, we encounter situations in programming where we need to make some decisions. Based on these decisions we execute the next block of code.

Flowcharts are very helpful in understanding and recognizing decision making structures. The following decision-making statements are available in Python.

- if-statements
- if...else statements
- if...elif...else statements
- nested if-statements

13.1. if-statement

If-statements consists of a Boolean expression which evaluates to TRUE or FALSE (see Figure 1). If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if-statement is executed. If the Boolean expression evaluates to FALSE, then the first set of code after the end of the if-statement(s) is executed.

![Figure 1: if-statement depicted in a flowchart](image.png)
Example 19 shows two situations of an if-statement code. In the first situation, the Boolean expression evaluates to TRUE and in the other situation to FALSE.

**Example**

```python
# Boolean evaluates to TRUE
x = 10
if x > 5:
    print("x is greater than 5")

# Boolean evaluates to FALSE
y = 10
if y > 100:
    print("y is greater than 100")
print("Bye bye!")
```

Example 19: if-statement in Python

### 13.2. if...else-statement

In the if...else-statement, the if-statement is followed by an optional else-statement. The if...else-statement contains a Boolean expression (see Figure 2) which if evaluates to FALSE, the else-statement is executed.

![Flowchart for if...else-statement](image)

Figure 2: if...else-statement depicted in a flowchart
Example 20 shows an if...else-statement code in which the Boolean expression is evaluated to FALSE which results in the execution of the else-statement.

**Example**

```python
x = 1
if x > 5:
    print("x is greater than 5")  # Boolean evaluates to TRUE
else:
    print("x is less than 5")    # Boolean evaluates to FALSE
```

Example 20: if...else-statement in Python

13.3. if...elif...else-statement

To check multiple expressions we use the *if...elif...else-statement*. *Elif* is short for *else if*. It is also called the *chained conditional statement* (see Figure 3). The condition of the next elif-statement is checked if the condition for the if-statement is FALSE. If the condition of the next elif-statement is FALSE, then the condition of the next elif-statement is checked and so on. If all the conditions are FALSE, the else-statement is executed. The if...elif...else-statement can have only one else-statement and multiple elif-statements. Note that only one elif statement is executed according to the condition.

![Flowchart](image.png)

Figure 3: if...elif...else-statement depicted in a flowchart
Example 21 shows an if...elif...else-statement code in which the condition of the elif-statement is evaluated to TRUE which results in the execution of the elif-statement.

Example

```python
x = 1
y = 5
if x > y:
    print("x is greater than y")
elif x < y:
    print("x is less than y")
else:
    print("x is equal to y")
```

Example 21: if...elif..else-statement in Python

### 13.4. Nested if-statement

Sometimes there may be situations in which you want to check for another condition after a condition evaluates to TRUE. In such situations, you can use nested if-statements. This is called *nesting* in computer programming. In nested if-statements, you can have an if...elif...else-statement inside another if...elif...else-statement. Nesting should be avoided when possible, as indentation is the only way to figure out the level of nesting in Python which can get confusing.

Example 22 shows a nesting code in Python in which the condition of the elif-statement is evaluated two times to TRUE before the program quits.

Example

```python
x = 10
if x < 11:
    print("x is less than 11")
    if x == 5:
        print("x is 5")
    elif x > 5:
        print("x is greater than 5")
    elif x < 5:
        print("x is less than 5")
else:
    print("Cannot guess x")
print("Leaving program")
```

Example 22: Nesting in Python
14. Loops
In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is met. The loop iteration technique is used to repeat the same or similar type of tasks based on a specified condition. Two of the most common types of loops are the while-loop and the for-loop. Besides the while-loops and the for-loops, we will also discuss here the nested loop.

14.1. While-loop
The while-loop is the simplest form of a programming loop. The statements in a while-loop are repeated as long as a given condition is TRUE (see Figure 4).

![Flowchart of a while-loop](image)

Figure 4: while-loop depicted in a flowchart

Example 23 shows loop created with the while-loop. Note that $x$ is incremented with one after every iteration, or else the loop will continue forever.
**Example**

```python
x = 1
while x < 5:
    print("Iteration: ", x)
    x = x + 1
```

Example 23: while-loop in Python

**Output example**

<table>
<thead>
<tr>
<th>Iteration: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration: 2</td>
</tr>
<tr>
<td>Iteration: 3</td>
</tr>
<tr>
<td>Iteration: 4</td>
</tr>
</tbody>
</table>

Table 30: Output of Example 23

### 14.2. For-loop

The for-loop is used when we want to repeat a piece of code ‘n’ times (see Figure 5) or when we want to iterate over a sequence (i.e., a list, a tuple or a string).

![Figure 5: for-loop depicted in a flowchart](image-url)
Example 24 shows how to repeat a piece of code ‘n’ times with a for-loop.

Example

```python
for x in range (0,4):
    print("Iteration:", x)
```

Example 24: repeat ‘n’ times with a for-loop in Python

Output example

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
```

Table 31: Output of Example 24

Example 25 shows how to iterate over a sequence with a for-loop in Python.

Example

```python
fruits = ["apple", "banana", "pineapple"]
for i in fruits:
    print(i)
```

Example 25: Iterate over a sequence with a for-loop in Python

Output example

```
apple
banana
pineapple
```

Table 32: Output of Example 25
14.3. Nested loop
We talk about nested loops when we use a loop inside another loop, e.g., a for-loop inside a while-loop or vice versa. We can also put a for-loop inside a for-loop and a while-loop inside a while-loop. Example 26 gives an example of a nested loop using a for-loop inside a for-loop.

**Example**

```python
for g in range(1, 3):
    for k in range(1, 3):
        print("%d * %d = %d" % (g, k, g*k))
```

*Example 26: A nested loop in Python*

**Output example**

1 * 1 = 1
1 * 2 = 2
2 * 1 = 2
2 * 2 = 4

*Table 33: Output of Example 26*

14.4. Loop control statements
A loop control statement is a statement that determines whether other statements will be executed or not. The Python language supports three loop control statements:

- break statement,
- continue statement, and
- pass statement.

14.4.1. Break statement
With the break statement we can stop the loop before it has looped through all the statements. It terminates the loop statement and transfers execution to the statement next to the loop. The break statement can be used in both while-loops and for-loops.

**Example**

```python
fruits = ["apple", "banana", "pineapple"]
for i in fruits:
    if i == "banana":
        break
    print(i)
```

*Example 27: break statement in Python*
14.4.2. **Continue statement**
With the continue statement we can stop the current iteration of the loop, and continue with the next. The continue statement can be used in both while-loops and for-loops.

**Example**

```python
fruits = ["apple", "banana", "pineapple"]
for i in fruits:
    if i == "banana":
        continue
    print(i)
```

**Output example**

apple
pineapple

Table 35: Output of Example 28

14.4.3. **Pass statement**
The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. Nothing will happen when the pass statement is used. It is therefore called a *null* operation. The pass statement is especially useful in places where your code will eventually go but has not been written yet.

**Example**

```python
for letter in 'Python':
    if letter == 'h':
        pass
    print("pass block")
    print(letter)
```

**Example 29: pass statement in Python**
Output example

```
pythont
pass block
h
on
```

Table 36: Output of Example 29
15. Functions

A function is a block of code which runs when it is called. Functions make codes reusable. We can pass data into a function and a function can return data as a result. The Python programming language has many built-in functions like print(), random(), and other built-in functions. It is also possible to create your own functions in Python. These functions are called *user-defined functions*.

15.1. Creating a function

In Python, a function block is started with the `def` keyword followed by the name of the function and parentheses. The code block within the function starts with a colon (:) and is indented (see Example 30).

**Example**

```python
def my_function():
    print("Hello World")
```

Example 30: Creating a function in Python

15.2. Calling a function

We can call a function just by using the function name followed by parenthesis (see Example 31).

**Example**

```python
def my_function():
    print("Hello World")
my_function()
```

Example 31: Calling a function in Python

15.3. Parameters

We pass data to functions as parameters which are specified after the function name inside the parenthesis. The parameters are separated with a comma, if the function has many parameters. Example 32 shows a function with one parameter, called name. When the function is called, it prints hello and the name that is passed along. The name that is passed along is used inside the function.
**Example**

```python
def my_function(name):
    print("Hello" + name)

my_function("John")
my_function("Catherine")
```

**Example 32: Parameters in Python**

Example 33 shows how to use a default parameter value in a Python function. When the function is called without a parameter, the default value is used.

**Example**

```python
def my_function(name = "World"):
    print("Hello" + name)

my_function("John")
my_function("Catherine")
my_function()
```

**Example 33: Default parameter value in Python**

### 15.4. Return values

With the `return` statement, we can let a function return a value (). This makes it possible to use a function output outside the function.

**Example**

```python
def average(x, y):
    avg = (x + y)/2
    print("Inside: Average is: ", avg)
    return avg

avg = average(2, 6)
# We can print avg outside the function
# because it is returned by the function
print("Outside: Average is: ", avg)
```

**Example 34: The return statement in Python**
15.5. Parameters vs arguments

The term parameter is often used to refer to the variable names within a function. An argument can be thought of as the value that is assigned to that variable. Parameters are in fact placeholders within a function for the arguments which are passed. In Example 35, `name` is the parameter for the function `my_function`. Anywhere we see `name` within the function will act as a placeholder until `name` is passed as an argument. The argument is passed to the function by `my_function("John")`. This calls the function, `my_function`, and assigns the value of 'John' (pass the argument 'John') to the parameter `name`. Now, the parameter `name` within the function will act as a variable with the value of 'John'.

```
Example

```
def my_function(name):
    print("Hello" + name)

my_function("John")
```

Example 35: Parameter vs argument
16. Modules
A module can be considered to be the same as a code library, a file consisting of Python code, e.g., set of functions, that we want to include in your application. It allows us to organize the Python code logically and makes the code easier to understand and use. Module codes can be reused as many times as we require.

16.1. Creating modules
We can create a module by saving the Python code in a file with the file extension .py. To create a module of Example 36, we just have to save the code in a file, e.g., my_module.py.

Example

```python
def greeting(name):
    print("Hello" + name)
```

Example 36: Creating a module in Python

16.2. Using modules

16.2.1. Import statement
We can use a module by using the import statement followed by the name of the module(s). The module names are separated by a comma when we import multiple modules. The module created in 16.1 can be used by importing the module named my_module and calling the greeting function (see Example 37). Use the syntax `module_name.function_name` when you use a function from a module.

Example

```python
import my_module
my_module.greeting("John")
```

Example 37: Using modules in Python

16.2.2. Import...from statement
With the import...from statement it is possible to import a part of a module instead of the whole module. In this way, we can import specific attributes from a module into the current namespace. Suppose we have a module, named my_module, as shown in Example 38.
def greeting(name):
    print("Hello" + name)

def gb(name):
    print("Goodbye" + name)

Example 38: Module with multiple functions

To import only the gb function from the module my_module, we use the syntax from module_name import name (see Example 39).

Example

from my_module import gb

gb("John")

Example 39: Import attributes from a module

Note that when import specific attributes from a module using the from keyword we do not use the module name when we refer to elements in the module, e.g., not my_module.gb("John"), but gb("John")

16.3. Variables in modules

Modules also can contain variables of all types, like arrays, dictionaries, objects, etc. (see Example 40 for a module named my_module).

Example

person1 = {
    "Name": "Catherine",
    "Age": "24",
    "Residence": "Amsterdam"
}

Example 40: Variables in modules

We can access the person1 dictionary by importing the module named my_module (see Example 41).
**Example**

```python
import my_module

country = my_module.person1["Residence"]
print(country)
```

*Example 41: Importing a dictionary from a module*

### 16.4. Renaming modules

We can create an alias for a module by using the `as` keyword when we import a module (see Example 42).

**Example**

```python
import my_module as MM

MM.greeting("John")
```

*Example 42: Renaming modules in Python*

### 16.5. Using the `dir()` function

The `dir()` built-in function returns a list of defined names belonging to a module (see Example 43). The list contains the names of all the modules, variables and functions that are defined in a module (see Table 37 for the output of Example 43). Note that the `dir()` function can be used on all modules, also on the ones we create ourselves.

**Example**

```python
import math

list_of_names = dir(math)
print(list_of_names)
```

*Example 43: Using the `dir()` function*
### Output example

```
['__doc__', '__name__', '__path__', 'acos', 'acosh', 'asin', 'asinh',
 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor', 'hypot', 'isinf',
 'isnan', 'log', 'log10', 'pi', 'pow', 'radians', 'sin', 'sinh',
 'sqrt', 'tan', 'tanh', 'trunc']
```

Table 37: Output of Example 43
17. **Date & Time**

There are several ways to handle date and time in a Python program. Python's time and calendar modules help track dates and times. Time intervals have to be represented in floating point numbers in units of seconds which are expressed by *tick* in python. The function `time()` from Python’s time module returns the current system time in ticks since 12:00 am, January 1, 1970 (epoch).

**Example**

```python
import time
ticks = time.time()
print("Number of ticks:", ticks)
```

*Example 44: Python’s time module*

**Output example**

```
Number of ticks: 1530196021.89
```

*Table 38: Output of Example 44*

Another way to work with time is using Python’s `datetime` module (see Example 45). The method `strftime()` is used to format date objects into readable strings. This method takes one parameter, `format`, to specify the format of the returned string. Table 40 shows a reference of all the legal format codes for the `strftime()` method.

**Example**

```python
from datetime import datetime
current = datetime.now()
print("Current date and time: ", current)
print("Today's date is: ", current.strftime("%Y-%m-%d"))
print("Year: ", current.year)
print("Month: ", current.month)
print("Day: ", current.day)
```

*Example 45: Python’s datetime module*
**Output example**

Current date and time: 2018-06-28 16:46:25.420000  
Today's date is: 2018-06-28  
Year: 2018  
Month: 6  
Day: 28

Table 39: Output of Example 45

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday, short version</td>
<td>Tue</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday, full version</td>
<td>Tuesday</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a number 0-6, 0 is Sunday</td>
<td>2</td>
</tr>
<tr>
<td>%d</td>
<td>Day of month 01-31</td>
<td>21</td>
</tr>
<tr>
<td>%b</td>
<td>Month name, short version</td>
<td>Nov</td>
</tr>
<tr>
<td>%B</td>
<td>Month name, full version</td>
<td>November</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a number 01-12</td>
<td>11</td>
</tr>
<tr>
<td>%Y</td>
<td>Year, short version, without century</td>
<td>17</td>
</tr>
<tr>
<td>%Y</td>
<td>Year, full version</td>
<td>2017</td>
</tr>
<tr>
<td>%H</td>
<td>Hour 00-23</td>
<td>13</td>
</tr>
<tr>
<td>%I</td>
<td>Hour 00-12</td>
<td>01</td>
</tr>
<tr>
<td>%p</td>
<td>AM/PM</td>
<td>PM</td>
</tr>
<tr>
<td>%M</td>
<td>Minute 00-59</td>
<td>31</td>
</tr>
<tr>
<td>%S</td>
<td>Second 00-59</td>
<td>09</td>
</tr>
<tr>
<td>%f</td>
<td>Microsecond 000000-999999</td>
<td>586512</td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset</td>
<td>+0100</td>
</tr>
<tr>
<td>%Z</td>
<td>Timezone</td>
<td>CST</td>
</tr>
<tr>
<td>%j</td>
<td>Day number of year 001-366</td>
<td>365</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of year, Sunday as the first day of week, 00-53</td>
<td>52</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of year, Monday as the first day of week, 00-53</td>
<td>52</td>
</tr>
<tr>
<td>%c</td>
<td>Local version of date and time</td>
<td>Tue Nov 21 13:31:00 2017</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>%x</td>
<td>Local version of date</td>
<td>11/21/17</td>
</tr>
<tr>
<td>%X</td>
<td>Local version of time</td>
<td>13:31:00</td>
</tr>
<tr>
<td>%%</td>
<td>A % character</td>
<td>%</td>
</tr>
</tbody>
</table>

Table 40: Reference of all the legal format codes for the strftime() method
18. Files I/O

Programs need an input to process and an output to display data (see Figure 6). The input can come, e.g., from a keyboard, an output from another program or a file. The output can be sent, e.g., to a computer screen, a printer, or another program.

![Input and output stream for a program](image)

We can store the data in variables while a program runs. But this data get lost if we terminate the program. If we want to keep the data after the termination of the program, we have to store the data in a file. The print function converts the expressions you pass into a string and writes the result to standard output ()

18.1. Writing to standard output

Writing to the standard output (stdout), e.g., a computer screen, can be done by the print statement.

*Example*

```python
print("Hello world")
```

*Example 46: Writing to standard output*

*Output example*

```
Hello world
```

*Table 41: Output of Example 46*
18.2. Reading from standard input
Reading from the standard input (stdin), e.g., a keyboard, can be done by Python’s built-in function input().

The input() function interpret the user’s input. If the user, e.g., puts in an integer value, the input function returns this integer value. If the user, on the other hand, inputs a list, the function will return a list. A prompt is displayed to enter the string.

Example

```python
str = input("Enter your input: ")
print("Received input is: ", str)
```

Example 47: The use of input() function to read from standard input

Output example

Received input is: Hello world

Table 42: Output of Example 47 if we give “Hello world” as input

18.3. Opening a file
We have to use Python’s built-in open() function to open a file before we can read or write a file. The open() function creates a file object and requires two arguments, the file name, and the file opening mode. Table 43 shows the different modes of opening a file in Python.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>This mode opens a file in text mode (default mode).</td>
</tr>
<tr>
<td>b</td>
<td>This mode opens a file in binary mode.</td>
</tr>
<tr>
<td>+</td>
<td>This mode opens a file for updating (reading and writing).</td>
</tr>
<tr>
<td>r</td>
<td>This mode opens a file for reading only (default mode).</td>
</tr>
<tr>
<td>rb</td>
<td>This mode opens a file for reading only in binary format.</td>
</tr>
<tr>
<td>r+</td>
<td>This mode opens a file for both reading and writing.</td>
</tr>
</tbody>
</table>
Table 43: Modes of opening a file in Python

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rb+</td>
<td>This mode opens a file for both reading and writing in binary format.</td>
</tr>
<tr>
<td>w</td>
<td>This mode opens a file for writing only. It overwrites the file if the file exists. If the file does not exist, it creates a new file for writing.</td>
</tr>
<tr>
<td>wb</td>
<td>This mode opens a file for writing only in binary format. It overwrites the file if the file exists. If the file does not exist, it creates a new file for writing in binary format.</td>
</tr>
<tr>
<td>w+</td>
<td>This mode opens a file for both writing and reading. It overwrites the file if the file exists. If the file does not exist, it creates a new file for writing and reading.</td>
</tr>
<tr>
<td>wb+</td>
<td>This mode opens a file for both writing and reading in binary format. It overwrites the file if the file exists. If the file does not exist, it creates a new file for writing and reading in binary format.</td>
</tr>
<tr>
<td>a</td>
<td>This mode opens a file for appending (pointer is at the end of the file). If the file does not exist, it creates a new file for writing.</td>
</tr>
<tr>
<td>ab</td>
<td>This mode opens a file for appending in binary format (pointer is at the end of the file). If the file does not exist, it creates a new file for writing in binary format.</td>
</tr>
<tr>
<td>a+</td>
<td>This mode opens a file for both appending and reading (pointer is at the end of the file). If the file does not exist, it creates a new file for reading and writing.</td>
</tr>
<tr>
<td>ab+</td>
<td>This mode opens a file for both appending and reading in binary format (pointer is at the end of the file). If the file does not exist, it creates a new file for reading and writing in binary format.</td>
</tr>
<tr>
<td>x</td>
<td>This mode creates a file. It returns an error if the file exists.</td>
</tr>
</tbody>
</table>

Table 44: Attributes related to file object

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.closed</td>
<td>This attribute returns <strong>TRUE</strong> if the file is closed and <strong>FALSE</strong> otherwise.</td>
</tr>
<tr>
<td>file.mode</td>
<td>This attribute returns the access mode with which file was opened.</td>
</tr>
<tr>
<td>file.name</td>
<td>This attribute returns name of the file.</td>
</tr>
</tbody>
</table>

We can get various information related to a file once the file is open by using attributes related to the file object. Table 44 shows the attributes which can be used with the file object.
Example 48 shows how we use the attributes on an open file.

Example

```python
demo = open("demo.txt", "w")
print ("Name of the file: ", demo.name)
print ("Opening mode: ", demo.mode)
print ("Closed: ", demo.closed)
```

Example 48: Opening a file and the use of the related attributes

Output example

```
Name of the file: demo.txt
Opening mode: w
Closed: False
```

Table 45: Output of Example 48

18.4. Closing a file

A file that is open can be closed by the `close()` method.

Example

```python
demo = open("demo.txt", "r")       # Open the demo file in reading mode
print ("Name of the file: ", demo.name)
demo.close()                       # Close the demo file
```

Example 49: Closing a file

18.5. Reading a file

We can read a file by the `read()` method once it is open by the `open()` function.

18.5.1. Read the whole file

Example 50 shows how we can read all the content of a file at once.

Example

```python
demo = open("demo.txt", "r")
# Read all the content of the demo file
str = demo.read()
print(str)
```

Example 50: Read all the content of the file
18.5.2. Read a part of a file
Example 51 shows how we can read a part of a file.

Example

demo = open("demo.txt", "r")
# Return the 5 first characters of the demo file
str = demo.read(5)
print(str)

Example 51: Read a part of a file

18.5.3. Read a few lines of a file
Example 52 shows how we can read one line of a file with the `readline()` method. If we want to read, e.g., the first three lines of a file, we call the readline() method three times.

Example

demo = open("demo.txt", "r")
# Read one line of the demo file
str = demo.readline()
print(str)

Example 52: Read one line of a file

18.5.4. Looping through a file
Example 53 shows how we can loop through a file line by line. This is useful when we want to read a whole file line by line.

Example

demo = open("demo.txt", "r")
# Looping through the file
for x in demo:
    print(x)

Example 53: Looping through a file line by line

18.6. Writing a file
With the `write()` method we can write to an open file. Note that the write() method does not add a newline character ('\n') to the end of a string.
Example 54 shows how we can add a new line to an existing file. The append mode ("a") adds strings to the end of a file.

```
Example

demo = open("demo.txt", "a")
demo.write("A new line is added to the file")
```

Example 54: Add a new line with the write() method

Example 55 shows how we can overwrite an existing file. The write mode ("w") overwrites any existing content.

```
Example

demo = open("demo.txt", "w")
demo.write("Existing content has been deleted")
```

Example 55: Overwriting an existing file

### 18.7. Creating a file

Three modes can be used to create a file with the open() method, namely with the modes create("x"), append("a"), and write("w").

- `create("x")`, it returns an error if the file exists.
- `append("a")`, file is only created if the specified file does not exist.
- `write("w")`, file is only created if the specified file does not exist.

Example 56 shows how a file named, demo.txt, is created if the file does not exist.

```
Example

demo = open("demo.txt", "w")
```

Example 56: Create the demo file

### 18.8. Deleting a file or a directory

We have to import the `os module` to delete a file or a directory. This module provides us methods to perform file processing operations, such as deleting and renaming files. To delete a file we use the `remove()` method. With the `rmdir()` method we can delete empty directories.
(folders). In Example 57 we check whether the file demo.txt exists or not. We delete the file if it exists otherwise we print “File not exist”.

Example

```python
import os

# Check if the file exists
if os.path.exists("demo.txt"):    os.remove("demo.txt")    print("File deleted")
else:    print("File does not exist")
```

Example 57: Delete a file

### 18.9. Directory methods

Example 58 shows some commonly used directory methods. We need to import the os module first in order to call the methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdir(path)</td>
<td>This method is used to change the current directory.</td>
<td>os.chdir(&quot;/misc/tmp&quot;)</td>
</tr>
<tr>
<td>getcwd()</td>
<td>This method displays the current working directory.</td>
<td>os.getcwd()</td>
</tr>
<tr>
<td>mkdir(path[,mode])</td>
<td>This method creates a directory.</td>
<td>os.mkdir(&quot;/misc/tmp&quot;)</td>
</tr>
<tr>
<td>rename(src, dst)</td>
<td>This method renames the file or directory src to dst.</td>
<td>os.rename(&quot;tut&quot;, &quot;tutorial&quot;)</td>
</tr>
<tr>
<td>rmdir(path)</td>
<td>This method deletes an empty directory.</td>
<td>os.rmdir(&quot;/misc/tmp&quot;)</td>
</tr>
</tbody>
</table>

Example 58: Some directory methods
Index

B
break statement ...................................................... 40

C
chained conditional statement ................................ 35
close() ...................................................................... 56
continue statement ................................................. 41

I
import statement .................................................... 46
import…from statement .......................................... 46
input ................................................................. 54

K
key-value pair .......................................................... 29

L
loop .......................................................................... 37

N
nested if-statements................................................ 36
nesting ..................................................................... 36
numerical data types ............................................... 15

O
open() ...................................................................... 54
Operand ..................................................................... 9
Operator .................................................................... 9
os module ............................................................. 58

P
pass statement ........................................................ 41
Python ........................................................................ 6
python interactive mode ........................................ 7
Python interpreter ................................................... 7
python script mode .................................................. 7

R
read() ....................................................................... 56
readline() ............................................................... 57
remove() .................................................................. 58
reserved keywords .................................................... 8
rmdir()...................................................................... 58

S
standard input ......................................................... 54
standard output ....................................................... 53
stdin .............................................. See standard input
stdout .................................................. See standard output

T
tick ........................................................................... 50

U
user-defined functions ............................................ 43

W
while loop ............................................................. 37
write() ...................................................................... 57