COMP122/20 - Data Structures and Algorithms

02 Python Fundamentals

Instructor : Ke Wei〖柯韋〗

➡ A319   ✆ Ext. 6452   ✉ wke@ipm.edu.mo

`http://brouwer.ipm.edu.mo/COMP122/20/`

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

January 6, 2020

Outline

1. **Interacting with Python**

2. **Python Syntax**

3. **Variable and Data Types**

4. **Iterables and Loops**

👁 *Textbook §1.1 – 1.5.*

Interacting with Python

IDLE

- IDLE is Python's Integrated DeveLopment Environment (IDE)
- The IDLE tool offers a more efficient platform to write your code and work interactively with Python.
- The Python Shell Window has dropdown menus and a >>> prompt. Here you can type and enter statements or expressions for evaluation.
- The Shell Window's editing menu allows you to scroll back to your previous commands, cut, copy, and paste previous statements and make modifications.
- The items on the File menu allows you to create a new file, open an old file, open a module, and/or save your session.
- In the File Window, you can write your Python code as a whole program.
- The File Window has the Run menu. When you choose to Run your code on the File Window, you can see the output on the Shell Window.

Keywords and Identifiers

- Python keywords have special meanings in Python, such as to denote statements and operations. The following are the keywords in Python.

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

- Keywords should *not* be used as identifiers, such as variable names, class names and function names.
- An identifier is a name given to a function, class, variable, module, or other objects to be used in a Python program.
- An identifier can be a combination of uppercase letters, lowercase letters, underscores, and digits (0-9). Digits cannot come first. The following are valid identifiers.

  myClass   my_variable   var_1   print_hello_world

- Python keywords and identifiers are *case-sensitive*. Thus, *Labor* and *labor* are different.

Statements and Multi-line Statements

- Statements are commands that a Python interpreter can execute.
- Statements include assignments, function calls, control flow statements and definitions.

```
a = 0                           def f(x):
print('Hello')                      return x**4   # the 4th power of x
for i in range(10):             class Foo:
    a += i                          pass
```

- The `pass` statement does nothing. It is used to fill an empty subclause when necessary.
- A statement may span over several lines. A line containing expressions inside parentheses, braces, and brackets can be broken at commas and operators.

```
ls = [1, 2, 3,          x = (10 * (a**3)       b = 10 < x < 20 \
      4, 5, 6]               - 5 * (a**2)           and y > 100 \
                            + 3 * a + 1)            or z < 0
```

- A backslash (\) at the end of every line indicates line continuation.

Indentation and Comments

- While most programming languages such as Java, C, and C++ use braces to denote blocks of code, Python programs are structured through indentation.
- A block of code can be identified when the statements start on the same column.

```
def car_rental_cost(days):
    cost = 35 * days
    if days >= 8:
        cost -= 70   # big discount
    elif days >= 3:
        cost -= 20   # small discount
    return cost
```

- If statements have to be more deeply nested, indent them further to the right.
- Characters on a line starting from a hash (#) symbol to the end of the line are comments.
- Multi-line comments are wrapped with triple quotes (''').

Variables

- A variable in Python is declared by assigning a value to it.

  my_number = 3
  my_string = 'ABC'

- There's no need to explicitly mention the type. The type of the value assigned becomes the type of the variable.
- The type(*x*) function tells the type of variable *x*.

  ```
  >>> type(my_number)                    >>> type(my_string)
  <class 'int'>                          <class 'str'>
  ```

- A variable can be set to a value of a different type later. Python is therefore a *dynamically-typed* language.

  my_number = [3, 4, 5]
  ```
                                         >>> type(my_number)
                                         <class 'list'>
  ```

Strings

- A string is a sequence of Unicode characters that may be a combination of letters, numbers, special symbols and even Chinese characters.
- In Python, a string is enclosed in matching single or double quotes.

  string1 = "It's␣double␣quoted."   *string2* = 'It␣is␣single␣"quoted."'

- The length of a string can be obtained by the len(*s*) function.
- A string can be indexed or subscripted, where indices start from 0.

  ```
  >>> len(string1)            >>> string1[6]                >>> string1[-3]
  19                          'o'                           'e'
  ```

- In Python, a string can be indexed by a negative number $-i$, meaning walking from the right. That is, $s[-i] = s[\text{len}(s) - i]$, provided $1 \leqslant i \leqslant \text{len}(s)$.
- Unlike in Java, indexing a Python string results a string with only one character, there's no character data type in Python.

Concatenating, Repeating and Slicing

- Strings can be concatenated together with the plus (+) operator.

  ```
  >>> 'Hello' + 'Python'                  >>> ('='+'AB'+'=')*4
  'HelloPython'                           '=AB==AB==AB==AB='
  ```

- It is easy to repeat strings with the times (∗) operator.
- Substrings are created with the slicing notation. There are two indices (separated by a colon) within square brackets. The first is the index to start the substring, the second is the index to stop.

  ```
  >>> '0123456'[2:5]          >>> string1[:4]                >>> string1[4:]
  '234'                       "It's"                         '␣double␣quoted.'
  ```

- If the start index is omitted, the slicing starts from 0, if the stop index is omitted, the slicing stops at the end.

The **range()** Function

- Python has an efficient way to handle a series of numbers and arithmetic progressions, by using the `range()` function.
- The `range(5)` returns an abstract collection of 5 elements, from 0 to 4, called an *iterable* (collection).
- An *iterator* is obtained from an iterable [by the `iter()` function] to return the elements one by one, by the `next()` function.

```
>>> iterable = range(5)
>>> iterator = iter(iterable)
>>> next(iterator), next(iterator), next(iterator), next(iterator), next(iterator)
(0, 1, 2, 3, 4)
```

- Iterators and calls to them are often performed implicitly.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

More Flavors of the **range()** Function

- Another flavor of `range(`*start*`, `*stop*`)` lets you specify the start/stop numbers.

```
>>> list(range(3, 8))
[3, 4, 5, 6, 7]
```

- A further *step* parameter, which is the increment, can also be specified by `range(`*start*`, `*stop*`, `*step*`)`. It can be a negative or positive number, but never zero.

```
>>> list(range(-2, 10, 2))              >>> list(range(2, -10, -3))
[-2, 0, 2, 4, 6, 8]                     [2, -1, -4, -7]
```

The **for** Loop

- The `for` loop iterates over an abstract list of items in an iterable collection.
- Results from the `range()` function and also lists are examples of iterables.

```
for p in [2, 3, 5, 7, 11, 13]:
    print(p*p)
```

- Use the `range()` function is efficient to loop through a series of numbers.

```
def locate_o(s):                        >>> locate_o('Hello Python')
    for i in range(len(s)):             "o" is @4.
        if s[i] == 'o':                 >>> locate_o('Bye-bye Java')
            print('"o" is @'+str(i)+'.')  "o" is not found.
            break
    else: print('"o" is not found.')
```

- The `break` statement ends the current loop and goes to the next statement after the loop, skipping over even the `else` clause, if any.