第 3 章

Python的基本概念

本章主要介绍 Python 程序设计的基本概念。

本章的学习目标：

- 掌握 Python 源程序的结构，编码、标识符、注释(含 docString)的使用；
- 掌握 Python 关键字、常见的内置函数、Python 名字空间的概念；
- 掌握 PYTHONPATH 环境变量的配置与使用、Python 模块与包的使用。

3.1 Python 相关的文件

3.1.1 Python 的几种文件类型

Python 系统软件既是一个 Python 程序开发环境，也是一个 Python 程序运行环境。Python 系统有以下几种类型的文件。

(1) py：Python 源代码文件。

(2) pyc：Python 字节码文件。

(3) pyw：Python 带用户界面的源代码文件。

(4) pyx：Python 包源文件。

(5) pyo：Python 优化后的字节码文件。

(6) pyd：Python 的库文件(Python 版 DLL)，在 Linux 上是 so 文件。

最常见的是 *.py 和 *.pyc 格式的 Python 程序文件。

3.1.2 Python 源程序示例

首先看一个 Python 源程序示例。该程序的功能是：打印当前日期时间，调用函数 hi()显示字符串"hello"＋参数信息。

【例 3.1】 Python 程序代码示例。

```
# - * - coding: utf - 8 - * -                    • 源文件编码格式声明
"""
Created on Tue Feb 25 09:07:47 2020
@author: Administrator                           • 注释信息
"""
import datetime                                  • 引入外部模块(库)

#获取当前时间                                     • 注释信息
t = datetime.datetime.now()                      • 源代码
def hi(name):                                    • 定义 get_py()函数
    """
    Parameters
    ----------
    f : list
        files and directories list               • 注释及帮助信息,可能用__doc__获取字符串内容
    Returns
    -------
    flpy : list
        list only contains python files.
    """
    return 'hello:' + str(name)                  • 函数返回值

if __name__ = = "__main__":                      • 判断是否运行在主模块
    print(t)
    print(hi('James'))                           • 程序源代码
```

将上面的程序代码保存到 c:\mycode\demo.py 文件。在操作系统提示符下输入 c:\python c:\mycode\demo.py,Python 会自动在其所在的目录下创建_pycache_文件夹,并将 demo.py 编译,得到 demo.cpython-38.pyc 字节码文件。程序运行结果:

```
2020 - 04 - 03 11:58:19.445626
hello:James
```

从上面的演示代码可以看出,一个 Python 程序通常由源码格式声明、注释、引入外部模块(库)、源代码、函数、类等语句组成。每条语句根据 Python 的语法规则构成,包含源码格式声明、注释、标识符、运算符、关键字、函数、类等。

3.1.3 Python 源程序编码格式

Python 源程序是文本文件,默认为 UTF-8 编码。可以使用任何文本编辑器编写 Python 源程序。不同操作系统下对文本编码可能不同,如在 Windows 操作系统下的记事本可以选择 ANSI、Unicode、Unicode big endian、UTF-8 类型的编码(见图 3.1)。这就导致 Python 的源程序可能存在多种编码。

图 3.1　源程序的编码格式

如果源码文件使用非 UTF-8 类型的编码，必须在 Python 源程序的第一行声明文件的编码格式。如：

```
# - * - coding: cp - 936 - * -
```

3.1.4　Python 源程序的注释与文档字符串

Python 有单行和多行注释两种方式。Python 程序的注释语句的作用主要是备注程序代码功能和逻辑关系、算法的编写思路，以便于程序的后期维护等。另外，符合规范的 Python 程序注释，可以自动生成对应的帮助文档。

(1) 单行注释。Python 中单行注释以 # 开头。实例如下：

【例 3.2】　第一个注释。

```
#第一个注释
print ("Hello, Python!")
```

(2) 多行注释。多行注释可以用多个 # 号或用成对的三个单引号 ''' 或三个双引号 """来标注。

例 3.1 所示的程序已经演示了单行注释与多行注释。

另外，Python 语言引入了文档字符串(docString)机制。文档字符串是一种多行注释，它作为模块、函数、类或方法定义中的第一条注释语句出现。这样的文档字符串自动成为该函数或对象的特殊属性__doc__。

__doc__属性可以被 Python 程序访问，而且还可以直接由相关工具生成 HTML 等格式的文档。Python 的 PEP257(https://www.python.org/dev/peps/pep-0257/)规定了文档字符串的标准格式。

【例 3.3】　查看__doc__属性。

通过代码：

```
print(hi.__doc__)
```

或者在 Python 提示符>>>下输入 help(hi),均打印如下信息:

```
Help on function hi in module __main__:
hi(name)
     Parameters
     ----------
    name : string
        a string,such as name
    Returns
    -------
    flpy : string
        hello + string
```

3.1.5 Python 语言的代码块

Python 语言的代码块是指具有一定逻辑功能的 n 行代码语句。Python 使用相同的缩进空格来表示同一代码块。

(1) 缩进空格。缩进的空格数是可变的,但是同级别的一个代码块的语句必须包含相同的缩进空格数。通常采用 4 个空格表示一个缩进。

【例 3.4】 Python 语言代码块缩进代码示例。

```
if True:
    print ("True")
else:
    print ("False")
```

如果同级别代码块缩进数的空格数不一致,会导致运行错误:

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
  print ("False")                                    #缩进不一致,会导致运行错误
  File "<tokenize>", line 6
    print ("False")
    ^
IndentationError: unindent does not match any outer indentation level
```

(2) 分行语句。Python 通常是一行写完一条语句,但如果语句很长,可以使用反斜杠(\)来实现将一行长的语句分成多行语句。

【例 3.5】 Python 语言一行长代码换行写法代码示例。

```
total = "A33"\
       "B44"\
       "C55"
print(total)
```

在[],{},或()中的多行语句,不需要使用反斜杠(\)。例如:

```
total = ['item_one', 'item_two', 'item_three',
        'item_four', 'item_five']
A33B44C55
```

3.2　Python 语言的关键字

Python 语言的关键字是构造 Python 逻辑程序代码的核心要素,关键字类似英语中的“单词”,它与用户定义的变量或函数组合构成程序语句代码。

关键字可以按类别分为常量、逻辑运算、程序流控制、异常与上下文处理、函数相关、模块与类管理 6 大类(见图 3.2)。

图 3.2　Python 语言关键字分类

另外,有趣的是 Python 提供了一个 keyword 模块,使用两行 Python 程序代码可以输出当前 Python 版本的所有关键字。

【例 3.6】 查看当前 Python 版本所有关键字。

```
>>> import keyword
>>> keyword.kwlist                    # 关键字列表
>>>       len(keyword.kwlist)          # 统计关键字的个数
```

上述代码运行结果:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def',
 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
35
```

可以看到,Python 共有 35 个关键字。有许多关键字与其他编程语言是相同的。如 'assert', 'break', 'class', 'continue', 'else', 'except', 'finally', 'for', 'if', 'import', 'return', 'try', 'while'等与 Java 语言的关键字是相同的,而且含义和用法也是一样的。

提示:程序员自定义自变量、函数不能与关键字相同,否则程序会报错。

3.3 Python 的标识符

Python 语言中用户定义的变量、函数、类名、模块等是用标识符来表达的。标识符有比较严格的命名规则,这些规则为:

(1) 标识符不能与关键字相同。

(2) 标识符的第一个字符必须是字母(ASCII 码字符或 Unicode 码字符)或下画线(_)。标识符对英文的大小写敏感。标识符的其他部分可以由字符、下画线、数字组成,标识符的长度没有限制。

可以使用非 ASCII 码字符做标识符,如中文作为变量名或函数名。如_ko、"中国"、disk 等是合法的标识符;3ks、in、+wps 等是不合法的标识符。

Python 中的特殊标识符为下画线标识符。

Python 有其专用的下画线标识符,"_"(1 个下画线)或 "__"(2 个下画线)可作为变量标识符的前缀和后缀来标识特殊变量。表 3.1 所示为 Python 中的下画线标识符。

表 3.1 Python 中的下画线标识符

类 型	描 述	含 义	示 例
_xxx	1 个下画线开始	保护变量,只有类对象自己和子类对象能访问到这些变量	_foo
__xxx	2 个下画线开始	私有成员,只有类对象自己能访问,连子类对象也不能访问	__foo
__xxx__	2 个下画线开头、结尾	通常是 Python 系统定义的名字	__init__()、__main__

一般来讲,变量名_xxx 被看作是"私有的",在模块或类外不可以使用。当变量是私有的时候,用_xxx 来表示变量是很好的习惯。

由于下画线对 Python 解释器有特殊的含义,建议初学者在不明确下画线的含义时,避免用下画线开头或结尾的标识符作为变量名。

3.4 Python 的内置常量

Python 内置的名字空间中已经定义了 6 个常量(见图 3.3)。

(1) False:bool 类型的假值。

(2) True:bool 类型的真值。

(3) None:NoneType 类型的唯一值。None 经常用于表示缺少值。

(4) Ellipsis:省略号文字,Python 使用 "…",主要与用户定义的容器数据类型的扩展切片语法结合使用。

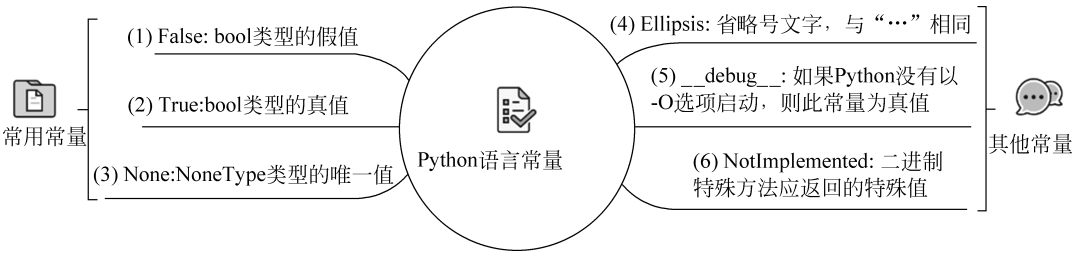

图 3.3　Python 语言中的 6 个常量

(5) __debug__：如果 Python 没有以 -O 选项启动，则此常量为真值。

(6) NotImplemented：二进制特殊方法应返回的特殊值(某些方法没有被实现，返回该错误)。

3.5　Python 的内置函数

Python 语言核心部分内置了大约 69 个内置函数(也称内部函数)。在 Python 程序中，可以直接调用这些函数(https://docs.python.org/3.9/library/functions.html)。进入 Python 系统，在>>>提示符下，输入 dir(__builtins__)，就会显示 Python 内置空间中的函数名、常量和关键字。

Python 语言内置函数可参看 https://docs.python.org/3.9/library/functions.html。可以将 Python 内置函数分为数字相关、数学运算、编码相关、序列相关、对象相关、系统函数、输入输出和变量相关 8 个大类(见图 3.4)。

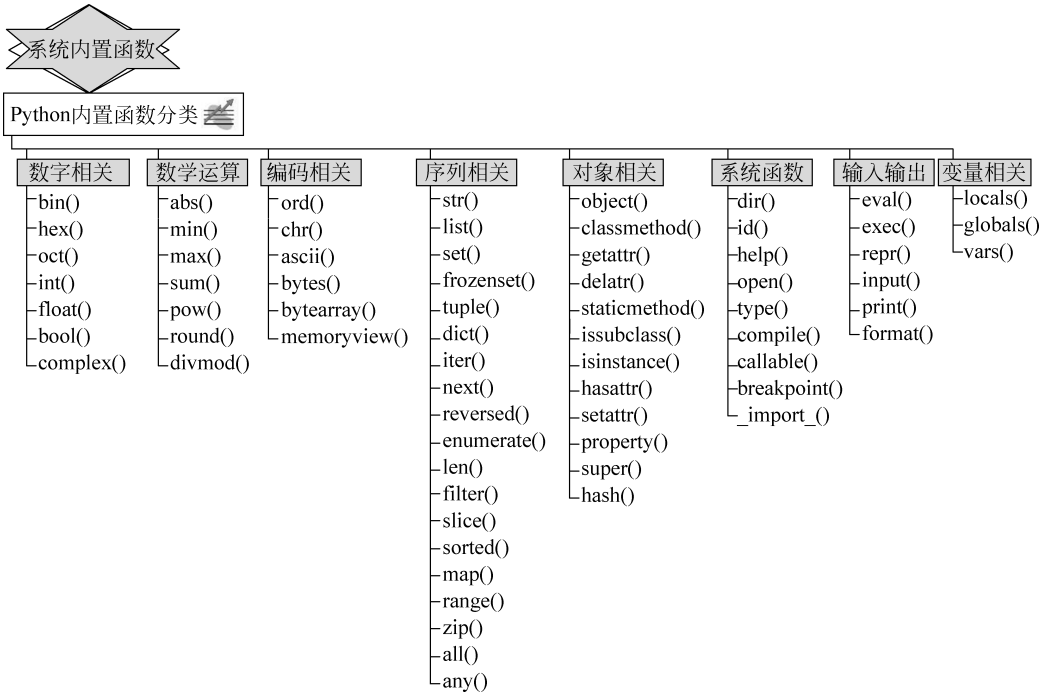

图 3.4　Python 语言内置函数及分类

提示：避免使用与内置函数名相同的标识符(变量名、函数、类名等)。否则，内置函数会被用户定义的同名标识符覆盖。

在此，先介绍 Python 使用频率最高的几个内置的函数(见图 3.5)。

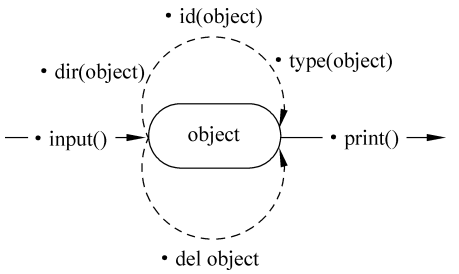

图 3.5　常见内置函数

(1) 对象相关的常用内置函数。

• id(object)

功能：返回对象在内存中的地址。

• dir(object)

功能：查看对象的相关名字空间，返回该空间的对象列表。

• type(object)

功能：可以获取 object 对象的类型。

• del object

功能：删除 object 对象。

(2) 常用输入输出函数。

• input([prompt])

功能：输入函数。显示提示信息，等待用户输入，返回输入字符串 str。

• print(* objects, sep=' ', end='\n', file=sys.stdout, flush=False)

功能：输出函数。将对象 objects 打印到输出流。

3.6　Python 的名字空间

在 Python 程序中，变量、函数、类等都是通过标识符来定义的。每一个标识符都会在相关的名字空间(namespace)占据一定位置。Python 会把命名后的变量、函数、类、对象分配到的相关的名字空间，并通过名称在相应名字空间中查找、使用它们。

Python 名字空间有两个作用：

(1) 区分不同作用域；

(2) 防止同名变量、函数、类等名字冲突。

若 Python 同一个名字空间中出现同名变量、函数、类等，则出现的同名变量、函数、类将覆盖先前的。

查看名字空间的命令函数是 dir()。

如，查看定义 x=100 前后 Python 名字空间内的变化情况。

【例 3.7】 查看名字空间。

定义变量前名字空间内的对象名称：

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

定义变量后名字空间内的对象名称：

```
>>> x = 100
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'x']
```

可以看到变量 x 已经占据了当前的名字空间的一个位置。

如果释放变量、函数、类等的名字空间，直接使用 del 加变量、函数、类模块名称。

【例 3.8】 del()函数案例。

清理、删除对象 x：

```
del x
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

可以看到运行了 del x 命令后，x 所占据的名字空间已经释放。

【例 3.9】 定义一个函数后的名字空间。

```
>>> def s(length):
    x = 300
    y = 400
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 's']
```

如图 3.6 所示，Python 有三个级别的名字空间，即局部、全局和内置名字空间。

图 3.6　Python 的名字空间及作用范围

(1) 局部名字空间。Python 会把在函数内部声明的变量放置到局部名字空间，记录函数内部的变量、传入函数的参数、嵌套函数等被命名的对象。

(2) 全局名字空间。Python 全局名字空间的变量其作用域为整个模块，记录模块 x 的变量、函数、类及其他导入的模块等被命名的对象。

(3) 内置名字空间。记录 Python 自身提供的内置函数、模块等被命名的对象。

可以使用 globals()、locals()、vars()来查看全局、局部变量的使用情况。

【例 3.10】 查看变量的使用情况。

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_
    importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__':
    <module 'builtins' (built-in)>, 'x': 100, 's': <function s at 0x0000000000445160>}
>>> locals()
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen
    _importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__':
    <module 'builtins' (built-in)>, 'x': 100, 's': <function s at 0x0000000000445160>}
>>> vars()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_
    importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__':
    <module 'builtins' (built-in)>, 'x': 100, 's': <function s at 0x0000000000445160>}
```

在 Python 程序运行过程中,会有局部名字空间、全局名字空间和内建名字空间同时存在。Python 对变量、函数、类的使用是按照“局部名字空间”→“全局名字空间”→“内置名字空间”的顺序查找的。如果找到了就使用,如果找不到,将放弃查找并引发一个 NameError 异常。

【例 3.11】 NameError 异常举例。

```
>>> aa
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    aa
NameError: name 'aa' is not defined
```

3.7 Python 的模块

一个 Python 源程序,如例 3.1 中的 c:\demo\demo.py,就是一个 Python 模块。如果要在其他程序中使用 demo.py 模块,首先要在环境变量 PYTHONPATH 中配置该模块的检索路径(见图 3.7)。

图 3.7 PYTHONPATH 的配置

然后,使用 import demo 命令,就可以使用 demo.py 模块中定义的函数、变量了。

【例 3.12】 导入模块。

```
>>> import demo
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
    '__spec__', 'demo']
```

【例 3.13】 查看 demo 名字空间。

```
>>> dir(demo)
['Info', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
      '__package__', '__spec__', 'datetime', 'hi', 't']
```

调用 demo 名字空间下的函数,必须在前面加名字空间名称。

【例 3.14】　调用模块内函数示例。

```
>>> demo.hi('james')
'hello:james'
>>> demo.Info()
hello from Info
<demo.Info object at 0x0000000002CA3F10>
>>>
```

提示：import demo 是将 demo.py 模块内的变量、函数导入 demo 名字空间下，使用时要加上 demo 名字空间名。

另一种导入方式是将 demo.py 模块内的变量、函数导入全局名字空间内。可以使用 from demo import * 语句直接调用函数名。

【例 3.15】　使用第二种导入方式调用函数示例。

```
>>> from demo import *
>>> dir()
['Info', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package
    __', '__spec__', 'datetime', 'hi', 't']
>>> hi('james')
'hello:james'
```

提示：from demo import * 是将 demo 模块内的变量、函数或类等导入全局名字空间内，可以直接使用相关函数。

3.8　Python 的包

Python 中的包(Package)是组织、管理源代码的一种方式。包就是一个容器，用来存放其他模块和子包。如，一个大的 Python 项目可能由不同开发人员分工完成，使用包来组织代码、管理代码可以避免出现模块名、变量名、函数名、类名等重名的问题。

Python 的包使用树状目录结构组织模块(见图 3.8)，将 Python 代码按照不同级别的目录存放，每个目录下面必须有个 __init__.py 文件(文件内容可以为空)，代表该目录下的 Python 文件使用包进行管理。

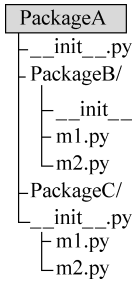

图 3.8　Python 中树状目录结构

可以简单地把包理解为按照不同级别的子目录来组织模块代码，每个目录下必须有一个 __init__.py 文件。包的本质就是一个含有 __init__.py 文件的文件夹。

- 使用 Python 包中的模块语法是通过目录的级别的方式来引用模块(如 PackageA.PackageB)。
- PackageB、PackageC 目录下均有同名 m1.py，m2.py 模块。这可以通过包名和模块名来区别的。如：

```
import Package.APackageB.m1
import Package.APackageC.m1
```

提示：Python 会在 sys.path 指定的目录(环境变量 PYTHONPATH 所指定的目录)下搜索包的路径。

包中的模块和函数，可以通过两种导入方式使用。

(1) import 包的名字。

(2) from 包的名字 import 函数名。

Python 系统用 import 导入包，会创建一个包的名字空间，包的名字来自__init__.py。

3.9 本章小结

图 3.9 所示为本章知识要点一览图。本章主要介绍了 Python 源程序的结构，编码、标识符、注释、Python 关键字、几个常见的内置函数、Python 模块和包。

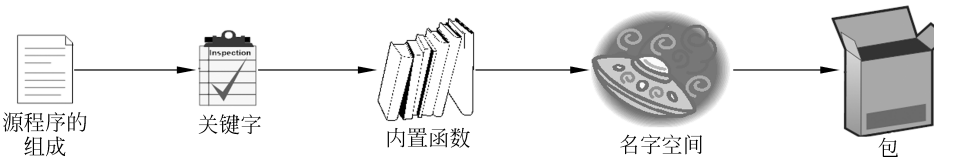

图 3.9 本章知识要点一览图


扫码观看

3.10 习题

(1) 编写一个打印字符串变量 s="hello"的程序 h.py。

(2) 利用 Python 的包管理机制，将 h.py 放在 a.b 包下，并利用 import a.b.h，输出 s 的内容。