

Python - P17

模块 (Module), 在 Python 中, 一个 .Py 文件即视为一个模块, 可以提高代码的可维护性
编写完成的模块可以在其他地方引用。Python 可以使用内置模块或第三方模块

内置函数

`abs(x)`, return the absolute value (绝对值) of a number (integer or float)

如果参数是复数 (complex number), 则返回复数的模 (modulus or magnitude)

`all(iterable)`, return False if any element of iterable is False, 等价于 `not any([not item for item in iterable])`

`any(iterable)`, return True if any element of iterable is True, 等价于 `any([item for item in iterable])`

注意: 当输入 iterable 为空, all() 返回 True, any() 返回 False

`ascii(object)` 与 `repr()` 类似, return a string containing a printable representation of an object

而 non-ASCII character 则用 \x, \u, \U 编码字符串

`bin(x)`, convert an integer to a binary string prefixed with '0b'

`hex(x)`, convert an integer to a lowercase hexadecimal string prefixed with '0x'

`oct(x)`, convert an integer to an octal string prefixed with '0o'

注意: 这三个函数的结果是合法的 Python 表达式 (valid Python expression)

如果输入不是 integer, 则尝试调用对象的 `_index_()` 来返回一个 integer

对于 prefix, 在正则表达式中也可以指定, 二进制 (%b), 八进制 (%o), 十六进制 (%x)

如 >>> '%#x' % 255, '%x' % 255, '%#1x' % 255 → '0xff', 'ff', '0xFF'

`class bool([x])`, return a Boolean value (True / False), converted by truth testing procedure

bool 是 int 的子类 (subclass), 但是不能作为其他类的父类, 只有两个实例 (instance) True / False

真值检查 (truth value testing), 任何 object 都可以用于 truth value testing

在默认情况下, object 会被当作 True.

除非 class 中定义了 `_bool_()` 方法返回 False 或是 `_len_()` 方法返回 0

Python 中的内置类型 (build-in object) 被认为是 False 的情形:

定义为 False 的常量 (constant defined to be False): None, False

数值类型的 0 值 (zero of any numeric type):

0, 0.0 (float), 0j (complex number), Decimal(0), Fraction(0, 1)

空的序列和集合 (empty sequences and collections):

'' (string), () (tuple), [] (list), {} (dict), set(), range(0)

Python - P18

ST Study
soft - circumscript

breakpoint(*args, **kws) drop into the debugger at the call site

调用时，从执行切换到调试器，且可以不选择 Python 自带的 pdb，而是任何设为首选调试器的调试器

在 Python 3.7 前，调试器必须手动设置，且调用代码冗长

而 Python 3.7 引入 breakpoint()，仅需一个调用命令，并且易于区分设置调试器和调用调试器

breakpoint() 本身是用 C 语言实现的，但可以用 Python pseudocode 描述实现过程

```
def breakpoint(*args, **kws):
```

实际上 breakpoint() 函数的作用

```
    import sys
```

仅是设置并调用钩子(hook)函数

```
    missing = object()
```

来源为 sys.breakpointhook

```
    hook = getattr(sys, 'breakpointhook', missing)
```

```
    if hook is missing:
```

如果 sys.breakpointhook 缺失

```
        raise RuntimeError('lost sys.breakpointhook')
```

则抛出 RuntimeError

```
    return hook(*args, **kws)
```

定义在 sys 模块中的 breakpointhook() 为 breakpoint 的钩子函数，定义了具体的功能实现

```
def breakpointhook(*args, **kws):
```

```
    import importlib, os, warnings
```

PYTHONBREAKPOINT 环境变量

```
    hookname = os.getenv('PYTHONBREAKPOINT')
```

其状态决定执行结果

```
    if hookname is None
```

完全不设置环境变量会默认进入 pdb 调试器

```
        or len(hookname) == 0:
```

为空字符串时与完全不设置环境变量效果相同

```
(PYTHONBREAKPOINT=)    hookname = 'pdb.set_trace'
```

注意 pdb.set_trace(*, header=None) 的参数

```
(PYTHONBREAKPOINT=0) elif hookname == '0':
```

为 '0' 时，立即返回，禁用调试

```
    return None
```

rpartition() 以指定字符(最右边)为界，将 str 拆成三段。

```
(PYTHONBREAKPOINT=)    modname, dot, funcname = hookname.rpartition('.')
```

若不存在，则全部在第三段

```
(callable)    if dot == ':':
```

为 'callable' 时，调用一个内置的可调用对象

```
    modname = 'builtins'
```

来源为 builtins 模块

```
try:
```

```
(PYTHONBREAKPOINT=some.importable callable)    module = importlib.import_module(modname)
```

模块导入失败或未找到

```
    hook = getattr(module, funcname)
```

调用对象，则 raise

```
except:    warnings.warn(RuntimeWarning)
```

```
    warnings.warn('Ignoring unimportable $PYTHONBREAKPOINT: {}'.format(hookname),
```

```
RuntimeWarning)
```

```
    return hook(*args, **kws)
```

返回钩子函数的调用

```
_breakpointhook_ = breakpointhook
```

如果 breakpointhook 被重写或缺失

可用 _breakpointhook_ 替换其重置

Python - P19

`class bytearray([source[, encoding[, errors]]])`, 返回一个新的字节数组, 且每个元素都在 $0 \leq x < 256$

注意 `bytearray` 是一个可变序列, 拥有可变序列的常用方法, 以及大多数字节类型的方法

根据 `source` 的类型不同, 函数又有不同的实现方式

如果没有输入任何参数, 即 `source` 为空,

则返回一个空的 `bytearray` 实例, 初始化的数组为空数组

即 `bytearray() → bytearray(b'')`

如果 `source` 为整数, 则返回一个长度为 `source` 的数组, 且数组元素为 '`\x00`'

即 `bytearray(13) → bytearray(b'\x00\x00\x00\x00')`

如果 `source` 为一个可迭代对象, 且不是字符串, 且所有元素为整型数 $0 \leq n < 256$,

则返回一个长度与可迭代对象相同长度的数组, 且其元素为可迭代对象的元素

即 `bytearray([1, 2, 3]) → bytearray(b'\x01\x02\x03')`

如果 `source` 是与 `buffer` 接口一致, 则此对象可被用于初始化 `bytearray`

if `source` is an object conforming to the `buffer` interface

initialize the `bytearray` with a read-only buffer of the object

如果 `source` 是字符串, 则必须传入 `encoding` 参数,

`bytearray()` 函数对字符串使用 `str.encode()` 方法,

使用传入的 `encoding` 参数将 `source` 转换为 `bytearray`

即 `bytearray('XYZ', 'utf-8') → bytearray(b'XYZ')`

如果没有 `encoding` 参数, 则返回 `TypeError: string argument without an encoding`

以及在 `string argument` 情形下不可传入可选的 `errors` 参数

`class bytes([source[, encoding[, errors]]])`, 返回一个新的字符串, 且每个位置的元素都在 $0 \leq x < 256$

注意 `bytes` 是一个不可变序列, 是 `bytearray` 的不可变的版本

拥有与 `bytearray` 相同的 non-mutating methods, 以及 indexing and slicing behavior

如 `bytes('XYZ', 'utf-8') → b'XYZ'`

`callable(object)` 如果传入 `object` 参数表现为 `callable`, 则返回 `True`, 否则返回 `False`

但是注意即使返回 `True`, 依旧可能引发 `call fail`, 但是返回 `False` 则必定失败

`class` 是 `callable`, 即 `i` 用 `class` 则返回一个新的实例 (`instance`), 即 `i` 用 `constructor`

而 `instance` 是 `callable` 当其 `class` 有 `_call_(self)` 方法

`(class call object):`

`def __init__(self):`

`def __call__(self, a, b):`

`pass`

`return a + b`

`c = call(), (1, 2) → 3`

Python - P20

129 - 2023.8



chr

(i), 用于将传入的整数参数 i, 转换为 unicode 下的字符串

如 `chr(97) → 'a'`, `chr(8364) → '€'`

注意, 合法参数取值范围是 0 至 0x10FFFF, 超出范围则抛出 ValueError

ord

(c), 用于将传入的 unicode 字符, 转换为字符对应的整数, 为 `chr()` 的逆函数

如 `ord('a') → 97`, `ord('€') → 8364`

len(str)

@classmethod 为一个函数装饰器(function decorator), 用于将一个 method 转化为 class method

class method 的第一个参数为表示自身类的 `cls` 参数 (implicit first argument)

即 `class_method(cls, arg1, arg2, ...)`, 与 `method(self, arg1, ...)` 中的 `self` 参数

class class_method(object):

 num = 1
] 定义类属性

 def __init__(self):
] 定义一个空的构造函数

 pass
] 未实现方法

 def func1(self):
] 定义 method, 为一个标准的类的方法

 print('call func1()')
] 必须经由类的实例调用

 @classmethod
] 定义一个 class method, 也称类方法

 def func2(cls):
] 传入参数 `cls`, 表示没有被实例化的类对象

 print('call func2()')
] 实例化 `cls` 并调用其中的 `func1()`

 print(cls)
] <class

 '__main__.class_method'>
] 定义方法

 cls().func1()
] 定义方法

 def func3():
] 定义 method, 但不传入 self 作为 implicit argument

 print('call func3()')
] 非标准的实例化方法

 print(class_method.num)
] so class_method.func2()

注意: class method 有两种调用方法, 一种是直接通过未实例化的类本身调用

另一种是通过类的实例调用, 如 `class_method()`.func2()

普通的 method 定义时使用 `self` 作为 implicit argument, 表示类的实例

所以如果直接经由未实例化的类本身调用, 如 `class_method().func1()`

则抛出 `TypeError: missing 1 required positional argument 'self'`

但是可以通过将实例传入作为参数调用, 如 `class_method().func1(class_method())`

非标准的 method 定义, 可以不使用 `self` 作为 implicit first argument

虽然也可以通过未实例化的类调用, 如 `class_method().func3()`

但如此调用的函数无法使用 `self` 调用实例的属性和方法

也无法使用 `cls` 调用类属性和类方法

另外注意, 类属性也可以通过未实例化的类直接调用, 如 `print(class_method.num)`

Python - P21

compile (source, filename, mode, flags=0, dont_inherit=False, optimize=-1)

将 source 转换为 code object 或 AST object

code object 可以用 exec() 或 eval() 执行

Source 参数传入的是用于转换的 source code, 可以是字符串, 二进制串 (byte string). AST 对象 filename 参数, 传入读取 source code 的来源文件

如果无需从文件读取, 则传入可读取值 (recognizable value), 通常为 <string>

mode 参数指定了代码的编译类型, 包括 'eval', 'single', 'exec'

'eval' 指一个单一表达式 (single expression)

'single' 指一个单一的指令语句 (single interactive statement)

'exec' 指一个指令序列 (sequence of statements)

flags 和 dont_inherit 是可选参数, 用来控制编译源码的标志

control which future statements affect the compilation of source

optimize 是可选参数, 指定 compiler 的优化级别 (optimization level of the compiler)

默认值为 -1, 等价于 interpreter 的优化等级中的 -O 选项

remove assert statements and any code conditional on the value of

其他参数 显式传入参数的值有: 0, no optimization, -debug- is true

1. asserts are removed, -debug- is false

2. both asserts and docstrings are removed

eval_code = compile('1 + 2 * 3 - 4', '', 'eval')] 输出为表达式

print(eval(eval_code)) → 3

exec_code = compile('print(1)\nprint(2)', '', 'exec')] 输出为指令序列

exec(exec_code) → 1\n2

single_code = compile('print(1)', '', 'single')] 输出为单一指令语句

exec(single_code) → 1

multi_code = compile('print(1)\nprint(2)', '', 'single')

→ SyntaxError: multiple statements found while compiling a single statement

即如果在单一语句模式下编译多语句, 会抛出 SyntaxError

但是 for 语句可以在单一语句模式下编译

for_code = compile('for i in range(3): print(i)\n', '', 'single')

exec(for_code) → 0\n1\n2

注意采用 multi-line code 时至少要有换行符, 否则抛出 SyntaxError: unexpected EOF while parsing

特别注意如 'for i in range(3):\n print(i)' 中没有正确缩进

会抛出错误: IndentationError: expected an indented block

注意如果 source 为 null, 则抛出 TypeError: compile() arg 1 must be a string, bytes, or AST object

Python - P22

class complex([real[, imag]])，返回一个值为 real + imag*j 的复数。
real 和 imag 都可以传入 int 和 float 类型的参数，且都可以为空。

如：传入 2 个值，complex(1, 2.3) → (1+2.3j)
传入 1 个值，complex(2.3) → (2.3+0j)，但是不能在不传入 real 时传入 imag

两个参数都 omitted，返回 0+0j，complex() → 0j，注意这里没有括号
允许向 real 传入一个字符串，并将字符串转换为复数。

如：complex('1+2.3j') → (1+2.3j)

但是注意，如果 real 传入了字符串，则 imag 必须为空

另外字符串必须符合 'real+imagj' 的形式，其中不可以存在空格

(如 complex('1 + 2.3j') → ValueError: complex() arg is a malformed string)

特别注意，real 和 imag 还可以传入 complex 类型的值

即对于如 $(1+1j)+(1+2j)j = -1+2j$ 的复数定义

可表示为 complex((1+1j), complex(1, 2)) → (-1+2j)

在 Python 中，Numeric Types 包含了 int, float 和 complex

用于支持 +, -, *, /, div, mod, 绝对值, 共轭, 幂等运算

在 Python 中，有一系列作用于对象属性 (object's attribute) 的函数

hasattr(object, name)：用于判断对象 object 中是否包含属性 name，其中 object 必须是对象，name 必须是字符串

如 class test(object): c = test(); 则 hasattr(c, 'p1') → True

p1 = 1; 则 hasattr(c, 'p2') → False

setattr(object, name, value)，用于将对象 object 中的属性 name 设置为 value；

如果对象中不存在属性 name，则 setattr 会为对象添加一个新的属性，名称为 name

如 setattr(c, 'p1', 2) → c.p1 = 2，之后 hasattr(c, 'p2') → True

setattr(c, 'p2', 3) → c.p2 = 3，在此之前 hasattr(c, 'p2') → False

getattr(object, name[, default])，用于返回对象 object 中属性 name 的值，如果没有属性 name 则返回 default

如 getattr(c, 'p2') → 3，这是反映了 getattr 是 setattr 的反函数 (counterpart)

getattr(c, 'p3', 4) → 4，c 中没有属性 p3，则返回 default 值 4

如果 object 中没有属性 name 也没有传入 default，则抛出 AttributeError

delattr(object, name)，从对象 object 中删除属性 name，delattr(object, name) 等价于 del object.name

如 delattr(c, 'p3') → 之后 hasattr(c, 'p3') → False，如果没有属性 name 则抛出 AttributeError

Python - P23

unit 21-23

`class dict()`, 用于创建一个新的字典(dictionary), 在 Python 中 `dict` 属于 ~~Mapping Types~~ Mapping Types
`dict()` 的 constructor 有多种传入参数的方式,

即不传入参数时, 默认地返回一个空字典 {}.

即只传入关键字参数, `dict()` 会使用这些关键字参数生成字典

如 `dict(one=1, two=2, three=3) → {'one': 1, 'two': 2, 'three': 3}`

利用传入的映射(mapping)和关键字参数生成字典

如 `dict({'one': 1, 'two': 2}, three=3) → {'one': 1, 'two': 2, 'three': 3}`

注意这里的映射是作为单一参数传入 `dict()` 函数后, 和将 `dict` 作为关键字参数传入不同

即 `dict(**{'one': 1, 'two': 2}, three=3) → {'one': 1, 'two': 2, 'three': 3}`

虽然结果相同, 但是这种属于传入关键字参数的情形

利用传入的可迭代对象 iterable 和关键字参数生成字典

如 `dict([('one', 1), ('two', 2)], three=3) → {'one': 1, 'two': 2, 'three': 3}`

注意 `dict((('one', 1), ('two', 2)), three=3)` 会得到相同结果

即 iterable 需要是类似于 $(\text{Foldable } t) \Rightarrow (\text{t } \xrightarrow{\text{a, b}})$ 的类型

另外 `dict` 可以与 `zip` 函数组合使用, 即直接从 [key] 到 [value] 生成字典

如 `dict(zip(['one', 'two', 'three'], [1, 2, 3])) → {'one': 1, 'two': 2, 'three': 3}`

`dir()`, 当不传入参数时, `dir()` 返回当前作用域(current local scope)中的名字的列表(list of names)

即包括当前作用域内已定义的变量、方法, 和类型名字的列表, 如 `['_Package__', '_spec__']`

如初始化后 `dir() → ['_annotations__', '_builtin__', '_doc__', '_loader__', '_name__', '_object__']`

返回传入的参数的属性、方法, 可以是模块、类或类的实例

首先, 如果参数中存在 `_dir_()` 方法, 则 `dir()` 函数会直接调用该方法

注意返回的结果会被视作一个 list

另外可以通过自定义 `_getattr_()` 方法的特殊行为来影响 `dir()` 函数返回的结果

另外 `dir()` 会通过对象的类型对象来获取信息(type object).

结果列表可能是不完全的, 如果有自定义的 `_getattr_()` 方法也可能是不准确的

另外 `dir()` 根据不同的传入参数类型有不同行为, 总会返回最相关的(relevant)而非最完全的(complete)

模块(module object), 返回模块中所有属性的名字

类型或类(type or class object), 返回其本身及其所有基类的属性的名字

实例(instance), 返回对象本身的, 其所属类型的及其所有基类的属性的名字

`divmod(a, b)`, 如果 a, b 是整数, 则返回商和余数的序偶, 即等价于 $(a // b, a \% b)$

如果 a, b 是浮点数, 则 $q = \text{math.floor}(a/b)$ 或减 1, 另外使得 $q * b + a \% b$ 非常接近于 a

其中 $a \% b$ 如果不为 0, 则 $a \% b$ 与 b 保持同号且 $0 \leq abs(a \% b) < abs(b)$