# 函 术

Python

清华大学计算机科学与技术系　乔　林

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

# Python 函 术

2

# Python 函 术

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

# Python 函 术

4

# 1. Python 函术的四项基本原则

5

1.1. 定义与调用
1.2. 数据有效性
1.3. 异常处理
1.4. 函数标注

# 1.1. 定义与调用

- 函数（function）：具有特定功能的可重用代码片段，实现解决某个特定问题的算法（algorithm）
  - 函数在需要时被调用，其代码被执行
  - 函数一般具有唯一的名称以供调用
  - 函数主要通过接口（interface）与外界通信，传递信息
- 目的与意义
  - 程序功能抽象，以支持代码重用
  - 使用时无需了解函数内部实现细节
  - 有助于采用分而治之的策略编写大型复杂程序

# 函数定义与调用的关键问题

- 主调函数与被调函数
  - 主调函数（*caller*，客户函数）：调用其他函数的函数
  - 被调函数（*callee*，服务器函数）：被其他函数调用的函数
- 关键问题：函数定义非函数调用
  - 函数定义并不执行函数体，函数体仅在函数调用时执行
  - 函数只能调用已实现的函数，但函数定义时可“调用”未实现的函数——函数定义顺序仅影响代码组织条理性

# 函数定义非调用，无需 *callee* 已完工

- 函数定义示例：双函数定义

```
def f():          # 定义函数 f() 可“调用”未实现的函数 g()
  print("In function f")
  g()             # 因为此“调用”在未真正调用函数 f() 时不会发生

def g():          # 定义函数 g()，确保其定义于函数 f() 实际调用之前
  print("In function g")

f()               # 实际函数调用，函数 f() 及其所需全部函数必须已实现
```

# 嵌套函数

- 嵌套函数：在函数内部定义函数
  - 内层函数仅供外层函数调用，外层函数之外不得调用
- 嵌套函数示例

```
def f():          # 外层函数 f()
   print("Outer function f")
   def g():       # 内层函数 g()，仅供 f() 内部调用，必须定义在实际调用前
      print("Inner function g")
   g()            # 函数 f() 调用内层函数 g()
# 若函数 g() 定义与调用顺序颠倒，则引发 UnboundLocalError 异常
f()               # 合法
f.g()             # 非法，不得调用嵌套函数，引发 AttributeError 异常
```

# Python函术的第一设计原则

- 言行一致
  - 函数定义（言）与函数调用（行）的格式必须完全匹配或兼容
  - *Trade War： To Triumph over Trump.....*

# 1.2. 数据有效性

- 程序容错
  - 重要数据的有效性与合法性是容错的关键节点之一
- Python存在的问题与解决方案
  - 问题：型的判定后置至运行期
  - 症结：动态型式语言的共性
  - 解决方案：数据有效性检查

# 素性判定示例

- 编写函数 *is_prime()*，判断正整数 *n* (*n*≥2) 素性。

```
def is_prime(n):
  import math
  if type(n) is not int:
    raise TypeError('must be an integer, but "%s" found' % type(n))
  if n < 2:
    raise ValueError('must be greater than 1, but "%d" found' % n)
  if n == 2:
    return True
  if n % 2 == 0:
    return False
  for i in range(3, math.ceil(math.sqrt(n)) + 1, 2):
    if n % i == 0:
      return False
  return True
```

# Python函术的第二设计原则

- 健壮性，健壮性，健壮性
  - 重要的事情说三遍.....
  - 要做身体健康的民族.....

# 1.3. 异常处理

- 异常（exception）定义
  - 程序中出现的很少见情况，是一种或各种例外
- 异常与错误
  - 异常可以是错误，错误也可以是异常
  - 异常不一定是错误，程序中可能出现的小概率事件也可以作为异常处理
  - 错误不一定是异常，部分错误无法处理，一旦出现此类错误，程序也无法恢复运行

# 异常处理逻辑

- 存在异常处理程序
  - 程序中存在的能够处理一类或各类异常的语句块
- 异常能够被引发
  - 出现某种特定情况时，Python 按照约定引发相关异常
  - 异常对象：Python 自动构造该类异常的一个对象，填充必要信息，并在引发异常时抛出
- 异常能够被捕获
  - 程序捕获该异常对象，流程转向异常处理程序，对该异常进行针对性处理；在此过程中，使用或不使用该异常对象均可

# finally 子句更重要！

- 使用 finally 子句确保资源被正确释放

```
# 引发 ZeroDivisionError 异常
# 且无论该异常有没有被处理，确保输出 "Oops..."
>>> try:
...     print(1 / 0)
... finally:
...     print("Oops...")
...
Oops...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

# 异常类的设计和组织

- Python 面向对象的异常类库层次架构
  - 全部异常类根类为 BaseException，类 Exception 为其派生类
  - Python 建议从类 Exception 而不是 BaseException 派生新的异常类
- 异常类型兼容性
  - 某个异常对象为某个异常类或其派生异常类的对象
  - 存在一个由异常类构成的类型元组，该异常对象与其中某个元素类型兼容

# 异常的引发与捕获

```
# 引发和重引发一个 RuntimeError 异常
>>> try:
...     raise RuntimeError("something no-good happened")
... except RuntimeError as e:
...     print(e.message)
...     raise
...
something no-good happened
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
RuntimeError: something no-good happened
```

# 异常链

```
>>> try:
...     print(1 / 0)
... except Exception as e:
...     raise RuntimeError("something no-good happened") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: something no-good happened
```

# Python函术的第三设计原则

- 危机管理
  - 小概率事件监控与资源控制
- 资源管理与控制极为重要
  - 有钱也不能瞎糟蹋，海外投资不能瞎买。要买油，买煤，买铁矿石，买铜，买钴，买.....

# 1.4. 函数标注

- 函数标注（annotation）目的
  - 提供用户自定义函数所使用的类型元信息
  - 示例：`def Add(x: int, y: int = 0) -> int: return x + y`
- 一种关注
  - 在 Python 3.6 中，函数标注可选，对函数其他部分无影响

# 整数输入的有效性检查

- 编写函数 *input_int()*，接受用户输入的整数，并检查返回数据的有效性。函数执行时应输出提示信息；若用户未提供该信息，函数应使用缺省信息提示用户。

# 整数输入的有效性检查

```
def input_int(prompt: str = None) -> int:
    t = input(prompt or "Please input an integer: ")
    while True:
# 尝试转换为整数，非整数时引发异常；处理该异常，请求用户重新输入
# 得到合法数据，return 语句跳出无限循环，返回合法整数
        try:
            return int(t)
        except ValueError:
            t = input("An integer needed. Try again: ")
m = input_int("Your age: ")
print(m)
n = input_int()
print(n)
```

# Python函术的第四设计原则

- 君子坦荡荡
  - 作为函数接口的一部分，函数标注十分必要
  - 有助于清晰地表达函数设计意图，指导函数使用者
  - 一带一路没有阴谋。要有，那也是天下大同的阳谋.....

# Python 函 术

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

Python 函 术

# 2. 名.象.量



2.1. 传统变量的四大基本特征

2.2. Python 变量与对象

2.3. 名空间与作用域

# 2.1. 传统变量的四大基本特征

- VANT
  - 值（value）
  - 地址（Address）
  - 名称（Name）
  - 型式（Type）
- 静态型式语言处理变量的逻辑
  - 编译期束定后三者，运行期束定值
  - 无 RTTI 时，运行期名称与型式信息缺失

- Python 数据对象的构造原则
  - 创建数据对象本身，而与其名称无关
  - 对象存续期：由 Python 自动管理
    - 典型实现策略：引用计数
  - 对象引用：名束定
    - 运行期束定名称与对象：必也正名乎？
- 一家之言
  - 量已过时，唯象当道

# 2.3. 名空间与作用域

- 名空间
- 作用域
- 全局量（对象）与局部量（对象）

# 名空间

- 名空间：从名称到对象的映射
  - 实现上，大多数名空间表现为符号表，以字典的形式组织，而象及其名存储于其中
  - *Python* 脚本中存在多个相互独立的名空间
- 名空间特性
  - 标识符独立性：不同名空间的同名标识符没有任何关联
  - 标识符唯一性：同一名空间中的标识符不得重名
  - 名空间嵌套：一个名空间可以包含另外一个名空间
    - 主要来源：嵌套函数定义、类成员函数

# 名空间分类

- 内置名空间（*built-in namespace*）
  - 内置名称集合，如内置函数名、内置异常名等
- 全局名空间
  - 模块内部的全局名称集合
- 局部名空间
  - 函数调用时的本地名称集合；对象的属性集合；嵌套函数的本地名称集合；类成员函数的本地名称集合
  - 不同函数调用使用不同名空间，如递归

# 名空间生存期

- 内置名空间
  - 创建时间：Python 解释器启动时
  - 删除时间：Python 解释器退出时
- 模块全局名空间
  - 创建时间：读入模块定义时
  - 删除时间：正常情况下，在 Python 解释器退出时
  - 注意事项：全局仅指在该模块内部为全局的；每个模块都有独立的全局名空间；导入模块后方可访问其中的全局标识符；模块中可能存在非全局标识符，即使导入模块也不可访问

# 名空间生存期

- 函数名空间
  - 创建时间：调用函数时
  - 删除时间：函数结束时，函数引发异常但却未在本函数内处理该异常时
- 类名空间
  - 创建时间：构造对象时
  - 删除时间：销毁对象时

# 作用域

- 作用域（*scope*）概念
  - 定义：可访问名空间中标识符的文法区域
  - 表现形式：在 *Python* 程序文本的某处，是否可以使用该名空间中的标识符
- 作用域与名空间的关系
  - 不在某名空间中，不能访问名该空间中的标识符
  - 在某名空间中，不一定能访问该名空间中的标识符
    - 主要原因：嵌套的名空间中的同名标识符可能导致名冲突

# 作用域分类与标识符查找

- 作用域分类
  - 局部作用域（最内层）：函数（类成员函数）、类、*Lambda*表达式形成的文法区域
  - 外层函数闭包作用域：嵌套函数的外层函数形成的文法区域
    - 注意：有可能存在多层嵌套
  - 全局作用域：模块形成的文法区域
    - 用户程序所在的模块为主模块 *__main__*
  - 内置作用域（最外层）：包含内置名称的文法区域
- 标识符查找顺序：由内向外

# 全局量与局部量

- 局部量
  - 定义于函数、类成员函数与 *Lambda* 表达式中的量
  - 类的数据属性：定义于类中，其表现与访问规则与局部量类似
  - 形式参数类似局部量，但因参数传递原因，有细微差异
- 全局量
  - 定义于类、函数、类成员函数与 *Lambda* 表达式之外的场合
- 名空间与作用域对量定义的影响
  - 全局量与局部量位于不同名空间，因而可重名
  - 发生重名时，局部量可能遮盖全局量的作用域，使其不可见

# 全局量与局部量示例

```
n = 42     # n 为全局量，位于全局名空间，其后代码（包括函数内部）均可访问

def double(x: int) -> int: # x 也为局部量，位于局部名空间，函数内可访问
  # 访问全局量 n
  print( "Before being doubled in double(): n = ", n )
  m = x * 2     # m 为局部量，位于局部名空间，函数内部可访问
  print( "After being doubled in double(): m = ", m )
  print( "After being doubled in double(): n = ", n )
  return m

print( "Before calling double() in __main__: n = ", n )
m = double(n)  # m 为全局量，位于全局名空间，与函数内部 m 为独立的两个对象
print( "After calling double() in __main__: m = ", m )
print( "After calling double() in __main__: n = ", n )
```

# 全局量与局部量示例

```
# 程序输出结果

# 调用函数前，全局量 n 值为 42
Before calling double() in __main__: n =  42
# 调用函数中，全局量 n 值为 42（加倍前）
Before being doubled in double(): n =  42
# 调用函数中，局部量 m 值为 84（加倍后）
After being doubled in double(): m =  84
# 调用函数中，全局量 n 值为 42（加倍后）
After being doubled in double(): n =  42
# 调用函数后，全局量 m 接受加倍值 84
After calling double() in __main__: m =  84
# 调用函数后，全局量 n 值维持 42 不变
After calling double() in __main__: n =  42
```

# 全局量与局部量示例

```
n = 42      # n 为全局量，位于全局名空间，其后代码（包括函数内部）均可访问
def double(x: int) -> int:
# 注释下一条语句，否则无法束定局部量 n，引发 UnboundLocalError 异常
# print( "Before being doubled in double(): n = ", n )
# 定义同名局部量 n（赋值即定义），新对象具有局部作用域，整个函数内部均有效
# 局部量 n 遮盖同名全局量 n 的部分作用域，使其不可见
# 局部量 n 定义前虽不能访问，但仍不允许上条注释语句访问全局量 n
# 换言之，即使前述注释语句出现在局部量 n 定义之前，n 也被解释为局部量
  n = x * 2
  print( "After being doubled in double(): n = ", n )
  return n
print( "Before calling double() in __main__: n = ", n )
m = double(n)
print( "After calling double() in __main__: m = ", m )
print( "After calling double() in __main__: n = ", n )
```

# 全局量与局部量示例

```
# 程序输出结果

# 调用函数前，全局量 n 值为 42
Before calling double() in __main__: n =  42
# 调用函数中，局部量 n 值为 42
After being doubled in double(): n =  84
# 调用函数中，全局量 m 接受加倍值 84
After calling double() in __main__: m =  84
# 调用函数后，全局量 n 维持原值 42 不变
# 即全局量 n 与局部量 n 虽同名，但不是同一对象
After calling double() in __main__: n =  42
```

- 问题：如何在函数内部修改全局量的值
  - 在函数内部可以引用全局量，但不能赋值——赋值将定义同名局部量
  - 进一步地，发生函数嵌套时，如何在内层函数中修改外层函数定义的局部量的值？
- 解决方案：量声明
  - 全局声明（global declaration）
  - 非局部声明（nonlocal declaration）

# 全局声明

- 全局声明：使用 *global* 关键字
  - 声明格式：*global identifier*
  - 多标识符全局声明：使用逗号分隔
  - 全局声明中的标识符不得为形式参数、不得位于 *for* 循环目标列表、类定义、函数定义、导入语句或变量标注中
- 含义：将其后标识符解释为全局的
  - 若该标识符在全局未定义，则此声明定义之，该全局量在函数调用结束后保持有效
- 有效性：在当前代码块各处均有效（包括声明之前）

# 全局声明示例

```
n = 42     # n 为全局量，位于全局名空间，其后代码（包括函数内部）均可访问

def double() -> int:      # 直接使用全局量 n，无需传递参数
  global n      # 声明全局量 n，函数内部对其赋值不会构造新的局部对象
  print( "Before being doubled in double(): n = ", n )
  n = n * 2     # 直接写入全局量
  print( "After being doubled in double(): n = ", n )
  return n

print( "Before calling double() in __main__: n = ", n )
m = double()
print( "After calling double() in __main__: m = ", m )
print( "After calling double() in __main__: n = ", n )
```

# 全局量与局部量示例

```
# 程序输出结果

# 调用函数前，全局量 n 值为 42
Before calling double() in __main__: n =  42
# 调用函数中，全局量 n 值为 42（加倍前）
Before being doubled in double(): n =  42
# 调用函数中，全局量 n 值更新为 84（加倍后）
After being doubled in double(): n =  84
# 调用函数后，全局量 m 接受加倍值 84
After calling double() in __main__: m =  84
# 调用函数后，全局量 n 维持更新后的值 84 不变
After calling double() in __main__: n =  84
```

# 非局部声明

- 非局部声明：使用 *nonlocal* 关键字
  - 声明格式：*nonlocal identifier*
  - 多标识符全局声明：使用逗号分隔
- 含义：将其后标识符解释为非局部非全局的
  - 从最内层嵌套名空间向外查找，一直到全局名空间（不含）
  - 若未找到该标识符，引发 *SyntaxError* 异常
- 有效性：在当前代码块各处均有效（包括声明之前）
- 非局部声明的用法类似全局声明

# Python 函 术

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

# 3. 函数参数与返回值

# 3.1. Python 函数参数传递机制为值传递吗？

- 形式参数（formal parameter/augment）与实际参数（actual parameter/augment）
  - 形式参数：函数定义时提供的参数
  - 实际参数：函数调用时提供的参数
- 参数传递（parameter passing）
  - 函数调用时，需要将实际参数传递给形式参数
  - 对象共享机制：Python 参数传递时不构造新数据对象，而是让形式参数和实际参数共享同一对象
  - 构造新对象时机：函数内部变更有常对象值时（写时复制）

# 参数传递机制示例

```
n = 2       # 在主模块 __main__ 中构造整型对象 2，将名称 n 束定于其上
# 定义函数 double()，加倍整数 x
def double(x: int) -> int:
# 函数调用时，名称 x 同样束定于整型对象 2 之上
# 函数内部进行表达式求值，需要重新设定 x 的值
    x *= 2                  # 2 为有常对象，值无法变更
# Python 构造一个新的整型对象 4，解除 x 的束定，重新束定其于新对象之上
    return x                # 返回新对象

# 调用函数 double() 时，实际参数 n 传递给形式参数 x，
# 使得函数 double() 内部的名称 x 同样束定于主模块的整型对象 2 之上，
# 此时并未构造一个新的整型对象
m = double(n)
```

# 参数传递机制示例

```
# 调用函数 id() 查看对象本征值
n = 2
print("id(n):", id(n))
def double(x: int) -> int:
  print("In double(): id(x):", id(x))
  x *= 2
  print("In double(): id(x):", id(x))
  return x
m = double(n)
print("m =", m)
print("id(m):", id(m))
print("n =", n)
print("id(n):", id(n))
```

# 参数传递机制示例

```
#脚本输出
id(n): 1381737600
In double(): id(x): 1381737600
In double(): id(x): 1381737664
m = 4
id(m): 1381737664
n = 2
id(n): 1381737600
```

# 3.2 位置参数与关键字参数

- 位置参数（positional parameter）
  - 位置固定：参数传递时按照形式参数定义顺序提供实际参数
  - 缺点：参数数目较多时，函数调用时容易混淆；缺省参数数目较多时，提供非缺省值时不能跳跃
- 关键字参数（keyword parameter）
  - 在函数调用时，提供实际参数对应的形式参数名称
  - 优点：明确标示实际参数和形式参数的对应关系；参数书写顺序更灵活
  - 缺点：增加函数调用时的代码书写量

# 关键字参数使用原则

- 关键字参数适用性
  - 普通参数和缺省参数均适用
- 关键字参数使用限制
  - 关键字参数必须位于位置参数之后
  - 关键字参数必须与函数定义时的形式参数名称匹配
  - 关键字参数顺序无限制
  - 不得重复提供实际参数

# 关键字参数使用示例

```
def f(x, y = 0, z = 0): pass     # 空语句，定义空函数体
f(1, 2, 3)                       # 合法
f(1, 2)                          # 合法，z使用缺省参数值
f(1)                             # 合法，y与z使用缺省参数值
f(1, , 3)                        # 非法，y为缺省参数，z也必须缺省
f(1, y = 2, z = 3)               # 合法，y与z为关键字参数
f(1, z = 3)                      # 合法，y使用缺省参数值
f(x = 1, y = 2, z = 3)           # 合法，均为关键字参数
f(z = 3, x = 1, y = 2)           # 合法，顺序无所谓
f(x = 1, z = 3)                  # 合法，x与z为关键字参数，y缺省
f(1, x = 1, z = 3)               # 非法，不得重复提供参数x
f(x = 1, 2)                      # 非法，关键字参数后不得有位置参数
f(1, y = 2, t = 3)               # 非法，关键字参数t不存在
```

# 3.3. 缺省参数值

- 缺省参数值（default augment value）定义格式
  - 函数定义：`def multiply(x, times = 2): return x *times`
  - 函数调用：`a = multiply(42)` 与 `b = multiply(42, 42)`
- 带缺省参数值的函数定义规范
  - 参数数目：无限制
  - 普通非缺省参数不得出现在此类参数之后
    - 函数定义时，必须普通非缺省参数在前，带缺省参数值的参数在后

# 带缺省参数值的函数调用规范

- 双参数调用函数 multiply()
  - 实际参数按照函数定义时的位置顺序分别传递给形式参数 x 与 times
- 单参数调用函数 multiply()
  - 实际参数传递给 x，times 使用缺省值 2
- 参数传递顺序
  - 当存在多个缺省参数值时，函数调用时必须按照位置顺序提供非缺省值，不能跳过部分缺省参数值

# 缺省参数值的意义

- 简化函数接口
  - 当函数部分参数大概率为某个固定值时，使用缺省参数值可以简化函数调用接口，提高编程效率和代码可理解性
- 一种关注
  - 缺省参数值技术可以部分实现函数重载（*overloading*）效果
    - 对于有缺省参数值的函数，其调用在外观上存在多个“不同”版本，具有一定的函数重载效果
    - 但是，这并不是真正的函数重载——*Python* 语言目前不支持函数重载，未来可能也不会支持——*Python* 中实际上有其他技术手段实现类似函数重载的功能

# 缺省参数值存在的问题与解决方案

- 缺省参数值的问题：高能预警！
  - 缺省参数值计算时机：只在函数定义时计算一次
  - 缺省参数值为表达式或无常对象时，如果其值在后续操作过程中发生变化，有可能导致不希望的结果
  - 示例：对于列表对象，下述代码导致列表元素累积

```
def f(a, ls = []):
    ls.append(a)
    return ls
print(f(1))          # 输出 [1]
print(f(2))          # 输出 [1, 2]
print(f(3))          # 输出 [1, 2, 3]
```

# 缺省参数值存在的问题与解决方案

- 对症下药
  - 原因分析：*ls* 的缺省值在函数定义时计算，构造一个空列表对象，其后所有函数调用都引用此列表对象，元素自然累积
  - 解决方案：使用 *None* 作为列表对象的缺省值，并在函数内部设置哨兵（*sentinel*），监视其变化

```
def f(a, ls = None):
    if ls is None:          # 本条 if 语句等价于 ls = ls or []
        ls = []             # 当 ls 为 None 时（函数调用时无第二个参数），
    ls.append(a)            # 真值测试为 False，构造空列表对象并赋值，
    return ls               # 否则引用原先的列表对象
```

# 3.4. 可变参数

- 可变参数：*def f(x, y, *args): ……*
  - 参数定义：**args* 表示形式参数 *args* 接受元素数目可变的元组作为实际参数
  - 适用场合：函数调用前不知道函数实际参数个数
- 注意事项
  - 可变参数必须出现在位置参数之后
  - 可变参数一般位于函数参数列表末尾，其后无其他参数或只有可变纯关键字参数

# 可变关键字参数

- 可变关键字参数：*def f(x, y, *args, **kwargs): ……*
  - 参数定义：***kwargs* 表示形式参数 *kwargs* 接受元素数目可变的字典作为实际参数
  - 适用场合：函数调用前不知道函数实际参数个数
- 注意事项
  - 可变关键字参数为纯关键字参数，不能以位置参数方式传入实际参数值
  - 可变关键字参数必须出现在参数列表末尾

# 可变关键字参数的适用性

- 特别说明
  - Python 2 与 Python 3 适用
    - def f(x, y, **kwargs): ……
    - def f(x, y, *args, **kwargs): ……
  - Python 3 适用：在函数头部列些全部纯关键字参数
    - 注意：纯关键字参数前必须存在可变参数标志“*[param_name],”，否则与位置参数无法区分
    - def f(x, y, *, error = True, verbose = False): ……
    - def f(x, y, *values, error = True, verbose = False): ……

# 可变关键字参数示例

- 编写函数，求数据序列均值。

```
def safe_mean(*values, **errors):
  error = errors.pop("error", True)          #获取关键字参数 error
  verbose = errors.pop("verbose", False)     #获取关键字参数 verbose
  if errors:                                 # 其他关键字参数非法
    raise TypeError("unexpected argument(s): {}".format(errors))
  try:                                       # 可变参数可能无法求值
    return sum(values) / len(values)
  except Exception as e:
    if verbose and error:                    # 显示详细异常链
      raise RuntimeError("error while invoking safe_mean()") from e
    elif error:                              # 仅显示系统异常
      raise
    else:                                    # 忽略异常，返回 None，可省略
      return None
```

# 可变关键字参数示例

```
# 验证代码
t = (1, 2, 3)
# 必须将元组t解包传递
print(safe_mean(*t))                                    # error = True, verbose = False
print(safe_mean(1, 2, 3))                               # error = True, verbose = False
# 以不同关键字参数调用
print(safe_mean(1, 2, 3, verbose = True)) # error = True
print(safe_mean(1, 2, 3, error = False, verbose = True))
print(safe_mean(1, 2, 3, error = False))   # verbose = False

# 脚本输出
2.0
2.0
2.0
2.0
2.0
```

# 可变关键字参数示例

```
# 验证代码
m = safe_mean(1, 2, "3", verbose = True)      # error = True
print(m)
# 脚本输出
Traceback (most recent call last):
  File "D:/Python_Programs/kwargs.py", line 16, in safe_mean
    return sum(values) / len(values)
TypeError: unsupported operand type(s) for +: 'int' and 'str'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "D:/Python_Programs/kwargs.py", line 26, in <module>
    m = safe_mean(1, 2, "3", verbose = True)
  File "D:/Python_Programs/kwargs.py", line 19, in safe_mean
    raise RuntimeError("error while invoking safe_mean()") from e
RuntimeError: error while invoking safe_mean()
```

# 可变关键字参数示例

```
# 验证代码
m = safe_mean(1, 2, "3")      # error = True, verbose = False
print(m)
# 脚本输出
Traceback (most recent call last):
  File "D:/Python_Programs/kwargs.py", line 28, in <module>
    m = safe_mean(1, 2, "3")
  File "D:/Python_Programs/kwargs.py", line 16, in safe_mean
    return sum(values) / len(values)
TypeError: unsupported operand type(s) for +: 'int' and 'str'

# 验证代码
m = safe_mean(1, 2, "3", error = False)    # verbose = False
print(m)
# 脚本输出
None
```

# 可变关键字参数示例

```
# 验证代码
m = safe_mean(error = True, verbose = True)
print(m)
# 脚本输出
Traceback (most recent call last):
  File "D:/Python_Programs/kwargs.py", line 16, in safe_mean
    return sum(values) / len(values)
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "D:/Python_Programs/kwargs.py", line 43, in <module>
    m = safe_mean(error = True, verbose = True)
  File "D:/Python_Programs/kwargs.py", line 19, in safe_mean
    raise RuntimeError("error while invoking safe_mean()") from e
RuntimeError: error while invoking safe_mean()
```

# 3.5. 返回值

- 函数返回逻辑
  - 函数执行时，控制流在遇到第一条 *return* 语句时，进行其后跟的表达式求值
  - 函数终止执行，将该结果作为返回值传递给主调函数
  - 主调函数可将返回的对象束定到其他名称上
    - 注意：非传统赋值，不实施实际的值复制行为
- 一种关注
  - 异常处理中的 *return* 语句：当 *return* 语句出现在 *try-finally* 语句的 *suite-try* 语句块中时，首先执行该函数的 *suite-fin* 语句块，函数随后才能终止执行

# 整数奇偶性示例

- 编写两个函数 *is_odd()* 和 *is_even()*，判断某个整数 *n* 是否为奇数和偶数。

```
def is_odd(n: int) -> bool:
  return n % 2 == 1
def is_even(n: int) -> bool:
  return n % 2 == 0

def is_odd(n: int) -> bool :
  return bool(n & 1)   # 需要显式转型
def is_even(n: int) -> bool :
  return not n & 1     # 已隐式（强制）转型
```

# Python 函 术

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

Python 函 术

# 4. 函数对象与 Lambda 表达式

# 4.1. 函数对象

- 函数定义语句可执行
  - 任务：在当前局部名空间（*local namespace*）或符号表（*symbol table*）中引入函数名称，束定函数名称与函数对象
- 程序设计辩证法：函数也是数据
  - 一条函数定义定义一个用户自定义函数对象
  - 访问函数对象：单独使用函数名称，无小括号
  - 函数对象（*function object, functor*）：封装函数可执行代码的打包器；含当前全局名空间（*global namespace*）的引用；函数执行时需要使用该全局名空间

# 函数对象的型与象

- 函数对象类型：函数对象的值类型为用户自定义函数
- 函数对象赋值：函数名称可赋给其他量，后者也可作为函数调用

```
>>> fib
<function fib at 0x1016b0b90>
>>> f = fib
>>> f
<function fib at 0x1016b0b90>
>>> f(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- 使用函数对象调用嵌套函数
  - 函数返回值可以为函数对象，通过该函数对象调用相应函数
  - 可返回嵌套定义的函数，从而在嵌套函数外部调用之

```
def f() -> function:    # 外层函数f()
  print("Outer function f")
  def g():              # 内层函数g()
    print("Inner function g")
  return g          # 外层函数f()返回内层函数g，f()为g()的包装函数
h = f()             # 调用函数f()，获得其返回的函数对象
h()                 # 合法，通过h执行内层函数g()代码
f()()               # 合法，省略量h定义
```

- 格式：`lambda [parameter_list]: expression`
  - 等价于如下定义：

```
def function_name(parameter_list):
    return expression
```

- Lambda 表达式目的与意义
  - 创建一个匿名函数，返回函数对象
  - 函数代码一般极短：用于一行表达式即可解决问题的场合
  - 适用于任何需要函数对象的场合

# Lambda 表达式示例

```
# 示例一
>>> f = lambda a, b: a + b
>>> type(f)
<class 'function'>
>>> f(1, 2)
3

# 示例二
>>> def inc(base):
...     return lambda x: base + x
...
>>> f = inc(42)          # 使用 f 引用 inc() 返回的函数对象，base 为 42
>>> f(42)                # 42 被累加到 base 上
84
```

# 字频统计示例

- 编写程序，统计文章中汉字出现频率。

第一章　青衫磊落险峰行

青光闪动，一柄青钢剑倏地刺出，指向在年汉子左肩，使剑少年不等招用老，腕抖剑斜，剑锋已削向那汉子右颈。那中年汉子竖剑挡格，铮的一声响，双剑相击，嗡嗡作声，震声未绝，双剑剑光霍霍，已拆了三招，中年汉子长剑猛地击落，直砍少年顶门。那少年避向右侧，左手剑诀一引，青钢剑疾刺那汉子大腿。

两人剑法迅捷，全力相搏。

练武厅东坐着二人。上首是个四十左右的中年道姑，铁青着脸，嘴唇紧闭。下首是个五十余岁的老者，右手捻着长须，神情甚是得意。两人的座位相距一丈有余，身后各站着二十余名男女弟子。西边一排椅子上坐着十余位宾客。东西双方的目光都集注于场中二人的角斗。

# 字频统计示例

```
# 读文本文件，生成字符串
def read_text(filename: str) -> str:
  with open(filename, "rt",
      encoding = "utf-8", errors = "ignore") as f:
    return f.read()
# 统计字频
def stat_char(text: str) -> list:
# 删除英文字母、数字、半角和全角标点符号
  for c in "abcdefghijklmnopqrstuvwxyz" \
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" \
      " ~!@#$%^&*()_+=-`[]\{}|:\";',./<>?，。、  " \
      "《》？；’：”“【】、｛｝§=-+——）（*&……%￥#@！～·\n":
    text = text.replace(c, "")
```

# 字频统计示例

```
# 构造字典，以字符为键，以其出现次数为值
    chars = {}
    for c in text:
# 若该字符为首次出现，增加键，构造字典，以字符为键，以其出现次数为值
        chars[c] = chars.get(c, 0) + 1
    result = list(chars.items())          # 由字典构造列表
# 以字符出现频率排序，降序
# key 为用户定义的回调函数，用于指定比较大小关系的数据对象
# sort() 函数执行时，传递待排序序列的元素给用户定义的 lambda 表达式
# lambda 表达式参数 x 接收到该元素
# 返回该元素（元组）的第 1 项（字频统计值）作为 sort() 函数比较依据
    result.sort(key = lambda x: x[1], reverse = True)
    return result
```

# 字频统计示例

```
# 验证脚本
filename = "金庸__天龙八部.txt"
text = read_text(filename)
conclusion = stat_char(text)
# 输出数据
# 注：因文本来自网络，未经认真检查，结果可能并不准确
print("总字数：", len(text))
print("用字数：", len(conclusion))
print("最常用百字频率如下：")
for i in range(100):
# conclusion[i]为元组，解包后传递给字符串的format()方法
  print("{}: {:>5d}".format(*conclusion[i]))
```

# 字频统计示例

```
# 《天龙八部》
总字数： 1253020
用字数： 4274
最常用百字频率如下：
不：20032
一：18686
的：18443
是：16485
道：15430
了：15289
人：12554
我：10971
你：10039
这：9906
他：9763
大：9054
来：8800
之：8008
说：7127
中：7059
在：6638
得：6595
下：6443
上：6416
子：6385
有：6232
那：6224
到：5577
段：5494
也：5486
手：5327
去：5058
便：4957
出：4921
心：4882
见：4562
么：4504
只：4489
个：4483
声：4343
自：4222
身：4193
然：3980
过：3740
无：3597
好：3580
时：3575
要：3519
誉：3490
又：3482
头：3372
着：3370
```

# 字频统计示例

| | | | |
|---|---|---|---|
| 她：3296 | 已：2710 | 但：2350 | 气：2041 |
| 老：3240 | 阿：2660 | 神：2297 | 十：2037 |
| 峰：3140 | 师：2658 | 都：2296 | 事：2018 |
| 想：3024 | 此：2643 | 生：2287 | 地：2017 |
| 知：2958 | 却：2643 | 叫：2283 | 死：2006 |
| 向：2956 | 起：2631 | 容：2257 | 女：1992 |
| 听：2930 | 什：2614 | 王：2236 | 少：1984 |
| 小：2911 | 和：2576 | 天：2221 | 武：1976 |
| 可：2906 | 们：2521 | 笑：2190 | 多：1956 |
| 如：2902 | 功：2481 | 正：2146 | 力：1936 |
| 当：2790 | 后：2464 | 三：2140 | 能：1926 |
| 将：2784 | 两：2393 | 萧：2100 | 前：1926 |
| 为：2777 | 而：2378 | 以：2082 | 竹：1926 |

# 字频统计示例

```
# 附加验证脚本
# 查看“爱恨情仇”出现频率
print(*[x for x in conclusion if x[0] in "爱恨情仇"])
# 具体结果如下——《天龙八部》中情仇是主线？
('情', 1074) ('仇', 647) ('爱', 349) ('恨', 222)
# 查看“兄哥弟姐妹”出现频率
print(*[x for x in conclusion if x[0] in "兄哥弟姐妹"])
# 具体结果如下——《天龙八部》中兄弟情谊是核心？
('弟', 1538) ('兄', 1148) ('哥', 1012) ('妹', 463) ('姐', 64)
# 查看“父母子儿女”出现频率
print(*[x for x in conclusion if x[0] in "父母子儿女"])
# 具体结果如下——《天龙八部》中母亲缺位，父子传承是关键？
('子', 6385) ('女', 1992) ('儿', 1770) ('父', 1393) ('母', 324)
```

# 字频统计示例

# 《笑傲江湖》
总字数： 994681
用字数： 3837
最常用百字频率如下：

不： 17464
一： 15549
道： 14253
的： 13825
是： 13415
了： 13215
人： 10572
我： 8703
这： 8472
他： 8336
你： 7759
令： 7204
大： 7095
来： 7022
之： 6963
狐： 6754
冲： 6636
说： 5847
在： 5819
得： 5784
上： 5547
有： 5515
中： 5366
子： 5351
剑： 5312
师： 5115
下： 4970
那： 4612
到： 4254
手： 4233
也： 4230
便： 3937
出： 3879
山： 3864
只： 3847
声： 3776
去： 3671
心： 3620
见： 3561
个： 3553
派： 3544
么： 3499
岳： 3384
然： 3336
好： 3215
自： 3180
又： 3077
们： 3005

# 字频统计示例

| | | | |
|---|---|---|---|
| 身：2984 | 门：2462 | 盈：2125 | 方：1754 |
| 过：2939 | 想：2461 | 却：2118 | 而：1738 |
| 可：2822 | 此：2428 | 老：2084 | 但：1731 |
| 甚：2810 | 笑：2410 | 后：2077 | 长：1726 |
| 向：2748 | 听：2350 | 她：2038 | 事：1719 |
| 头：2696 | 如：2337 | 法：2014 | 对：1716 |
| 无：2691 | 知：2324 | 生：1985 | 多：1706 |
| 当：2667 | 小：2311 | 天：1978 | 三：1696 |
| 时：2648 | 将：2305 | 两：1938 | 十：1688 |
| 要：2625 | 弟：2297 | 都：1934 | 前：1685 |
| 和：2616 | 为：2234 | 教：1893 | 群：1681 |
| 林：2510 | 已：2185 | 叫：1835 | 没：1628 |
| 着：2492 | 起：2139 | 气：1759 | 招：1616 |

# 字频统计示例

```
# 《笑傲江湖》
# 查看“爱恨情仇”出现频率
print(*[x for x in conclusion if x[0] in "爱恨情仇"])
# 具体结果如下——与《天龙八部》顺序一致
('情', 840) ('仇', 274) ('爱', 228) ('恨', 111)
# 查看“兄哥弟姐妹”出现频率
print(*[x for x in conclusion if x[0] in "兄哥弟姐妹"])
# 具体结果如下——与《天龙八部》顺序一致
('弟', 2297) ('兄', 1263) ('哥', 1038) ('妹', 674) ('姐', 170)
# 查看“父母子儿女”出现频率
print(*[x for x in conclusion if x[0] in "父母子儿女"])
# 具体结果如下——与《天龙八部》仅“父女”对调
('子', 5351) ('父', 1321) ('儿', 1284) ('女', 1080) ('母', 139)
```

# Python 函 术

1. Python 函术的四项基本原则
2. 名.象.量
3. 函数参数与返回值
4. 函数对象与 Lambda 表达式
5. 总　结

Python 函 术

# 5. 总　结

# 5. 总 结

- Python 是事务处理型语言，函数的作用与地位比其他语言更重要
- Python 函数定义规则十分复杂，学好、用好并不容易

# Python 函 术

# 谢　谢