python作为 [redacted]

作为 [redacted]

[redacted]

• Basic

print `print("string")` `print("Hello"+"World")`

于 串 [redacted] `print("Let\'s go! ")`

`\`

于 [redacted] `\n` `print("Hello!\nHi!")`

```
print('''Is this the real life?
Is this just fantasy?
Caught in a landslide,
no escape from reality''')
```

```
math_now = 120
print("     math_now     " + math_now )
print("=====")
math_now = math_ex
print("     math_ex     " + math_ex )
print("     math_now     " + math_now )
math_now = 150
print("=====")
math_now = 100
print("     math_now     " + math_now )
```

上 一个变
变 值.

> 变量命名不能有空格，不能是数字开头
>
> 变量应该尽量理解和记忆，一般用英文

```
greet = "        "
print(greet + "  三")
```

python在3.0 [redacted]

python命名 [redacted]

变 名[redacted]关

计算

单 与C [redacted]

```
import math
math.sin(1)
print(math.sin(1))
```

注释 先 个

> When I wrote this code,only God & I understood what it did.
>
> Now...only God knows.

作 人

```
#
"""


"""
```

数据类型

```
"Hello" #       str
6   -32 #     int
6.0 10.07   #       float
True False  #         bool
None    #         NoneType
#           etc.
```

串 函 为`len`.

```
"Hello"[3]
#                           \n
#              C
```

`type` 函 可以 回 型 型决 了你 上 什么函 .

交互模式 python两 命令 和交互 。

交互 可以 在 制台 cmd 写一 一 .

input

```
input("       ")
food = input("       ")
print(food)
# input
#
# int, float, str,
food_num = int(input("         "))
food_per = food_num / 1
print("       " + str(food_per) + "  ")
```

• 条件判断

为`==`，不　于号为`!=`

下　们　出一个例

```
# BMI
# BMI =　　/　　** 2

#　　BMI
user_weight = float(input("　　(　　: kg) : "))
user_height = float(input("　　(　　: m) : "))
user_BMI = user_weight / (user_height) ** 2
print("　BMI　为: " + str(user_BMI))

#　　BMI
if user_BMI <= 18.5:
    print("　BMI　　")
elif 18.5 < user_BMI <= 25:
    print("　BMI　　")
elif 25 < user_BMI <= 30:
    print("　BMI　　")
else:
    print("　BMI　　")
```

在判　　况下可以使　与　　代

```
and or not
# not
```

• List

分与C　　列　和其　型不同　在于列　可变　。

python　列　可以　不同　列　于　元　，　们假　list　一个列

`list.append`和`list.remove`　从列　中加减

```
shopping_list = []
shopping_list.append("　")
shopping_list.append("　")
shopping_list.remove("　")
shopping_list.append("　")
shopping_list.append("　　")
shopping_list[1] = "　"
print(shopping_list)
print(len(shopping_list))
print(shopping_list[0])

price = [799, 1024, 200, 800]
max_price = max(price)
min_price = min(price)
```

```
sorted_price = sorted(price)
print(max_price)
print(min_price)
print(sorted_price)
```

于列　加减元　　受　C　　单

• Dictionary

典　　值　key and value　key　　值

key一　不　修　　以　key　　个元　可以使　tuple　元

典可变　但　和元　不可变。

key　否在　典中可以　`"key" in dictionary`, key　　print后会　回　值

```
#    input      if
slang_dict = {"YYDS":"“      ”      ,                    "}
#
slang_dict["    "] = "                    业              "
slang_dict["    "] = "                        上          ,            "
slang_dict["     "] = "                       "
slang_dict["    "] = "                ,          ,      , 不        “   ”
          ,     不                    “   ” 为       ,            "
slang_dict["   不 ,          "] = "                    不       "

query = input("                  : ")
if query in slang_dict:
    print("       " + query + "     下")
    print(slang_dict[query])
else:
    print("                    ")
    print("                为: " + str(len(slang_dict)) + "    ")
```

其中关于　典　　三个　作

```
dict.keys()     #
dict.values()   #
dict.items()    #
```

• 循环

Python　　两　一　for...in　　依　list　tuple中　个元　代出

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)
```

以`for x in ...`　　个元　代入变　`x`　后　　块　句。

再　　们　　1-10　　之和　可以　一个`sum`变　做　加

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

　　1-100　　之和　从1写到100　　困　　Python　供一个`range()`函　　可以　　一个　　列　再`list()`函　可以　　为list。　`range(5)`　　列　从0　　于5

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

`range(101)`　可以　　0-100　　列　　下

```
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

　二　　while　　只　件　　不　　件不　　出　。　　们　　100以内　　之和　可以　while

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
```

在　　内　变　`n`不　　减　　到变为`-1`　　不再　　while　件　　出。

- **格式化字符串**

Part 2 函数

　义函　　函　名和参　个　　可以先　参　　型做

函　体内　可以　`return`　　回函　　函　　也　`return`　句　　动`return None`。

函　可以同　　回　个值　但其　　一个tuple。

　们　　举个例

```
def calculate_magnetic_field(current_input):
    magnetic_field = current_input * 3.0144
    return round(magnetic_field, 1)

def calculate_resistance(current_reading):
    resistance = (4 / (current_reading*1000)) * (10 ** 6)
    return round(resistance, 1)
```

```
def main():
    current_input = float(input("                : "))
    current_reading = float(input("              : "))
    magnetic_field = calculate_magnetic_field(current_input)
    resistance = calculate_resistance(current_reading)
    print(f"      : {resistance} Ω")
    print(f"        : {magnetic_field} Gs")

if __name__ == "__main__":
    main()
```

`if __name__ == '__main__'` 代 作 **判断当前模块是否是直接运行** 。

你 一个 Python

- 你 关 个 `__name__` `'__main__'` 会亮 代 块 内 。
- 个 到其他 件 入 `__name__` 变 块名 不会亮 不 代 块 内 。

作

1. **直接运行时** 会 个代 块 像 入口
2. **被 入时** 不会 代 只 入其他 分

• 引入模块

Part 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

> 面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。
>
> 而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。
>
> 在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

• 类和 例

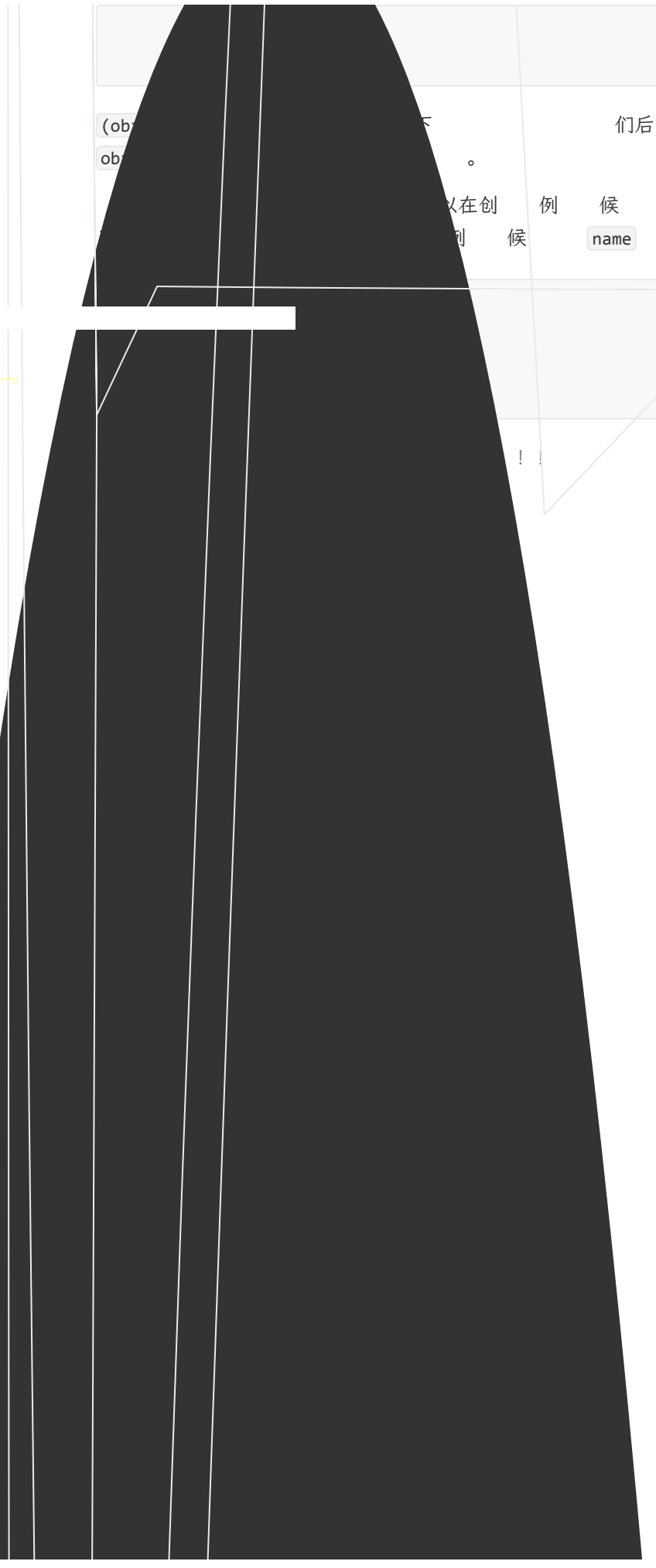
《 世 》

> 为什么每一匹马都是一样的?
>
> 点心师该如何做出五十个一模一样的姜饼人?

上 图
人 下 一个

以

们以 "Student" 举例 `class`



```
class Student(object):
    ...

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同 `get_grade` 可以 在 例变 上 不 内

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

lisa = Student('Lisa', 99)
bart = Student('Bart', 59)
print(lisa.name, lisa.get_grade())
print(bart.name, bart.get_grade())
```

后 出一个 习 上

```
#
#
# 1.
# 2.
# 3.

class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self.grades = {"    ": 0, "    ": 0, "    ": 0}

    def set_grade(self, course, grade):
        if course in self.grades:
            self.grades[course] = grade

    def print_grades(self):
        print(f"    {self.name} (    : {self.student_id})      为: ")
        for course in self.grades:
```

```
            print(f"{course}: {self.grades[course]}  ")

chen = Student("    ", "100618")
chen.set_grade("    ", 92)
chen.set_grade("    ", 94)
chen.print_grades()
# zeng = Student("    ", "100622")
# print(chen.name)
# zeng.set_grade("    ", 95)
# print(zeng.grades)
```

- **继承**

们仍 人作为例 。

人作为 一 们可以 人具 一些共 —— 在 义 后可以 人 个
些 加上一些 —— python 。

们 写了一个名为`Animal` class 一个`run()` 可以 印

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

们 写`Dog`和`Cat` 可以 从`Animal`

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

于`Dog` `Animal` 于`Animal` `Dog` 。`Cat`和`Dog` 似。

什么 了 全 功 。 于`Animal` 了`run()` 因 `Dog`和`Cat`作为
什么事也 动 了`run()`

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

下

```
Animal is running...
Animal is running...
```

也可以 加一些 Dog

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

二个 们 代 做一 。你 到了 `Dog` `Cat` 们`run()` 候 `Animal is running...` 合 做 分别 `Dog is running...`和`Cat is running...` 因 `Dog`和 `Cat` 下

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')
```

和 在 同 `run()` 们 `run()` 了 `run()` 在代 候 会 `run()`。 们 了 另一个 。

后 出一个 习

```
#
# -                    FullTimeEmployee          PartTimeEmployee
# -              "     name"  "     id"
#       备"        print_info"
# -       "     monthly_salary"
#         "     daily_salary"      "            work_days"
# -               "        calculate_monthly_pay"

class Employee:
    def __init__(self, name, id):
        self.name = name
        self.id = id

    def print_info(self):
        print(f"       : {self.name},     : {self.id}")

class FullTimeEmployee(Employee):
    def __init__(self, name, id, monthly_salary):
        super().__init__(name, id)
        self.monthly_salary = monthly_salary

    def calculate_monthly_pay(self):
        return self.monthly_salary

class PartTimeEmployee(Employee):
    def __init__(self, name, id, daily_salary, work_days):
        super().__init__(name, id)
        self.daily_salary = daily_salary
```

```
        self.work_days = work_days

    def calculate_monthly_pay(self):
        return self.daily_salary * self.work_days

zhangsan = FullTimeEmployee("  三", "1001", 6000)
lisi = PartTimeEmployee("    ", "1002", 230, 15)
zhangsan.print_info()
lisi.print_info()
print(zhangsan.calculate_monthly_pay())
print(lisi.calculate_monthly_pay())
```

Part 4 文件

- read

```
#          data.txt
#

# 1. read
with open("./data.txt", "r", encoding="utf-8") as f:
    content = f.read()
    print(content)

# 2. readline
with open("./data.txt", "r", encoding="utf-8") as f:
    line = f.readline()
    while line != "":
        print(line)
        line = f.readline()

# 3. readlines
with open("./data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()
    for line in lines:
        print(line)
```

- write

```
#     1                "poem.txt"
#
#
#
with open("./poem.txt", "w", encoding="utf-8") as f:
    f.write("           , \n           , \n    不      \n")

#     2         "poem.txt"
#
#
with open("./poem.txt", "a", encoding="utf-8") as f:
    f.write("         , \n")
    f.write("           ")
```

Part 5 测试

先 出一个 例   内 后   充

```
'''
    :            下
                 下    shopping_list.py,        下          :

class ShoppingList:
    """             , shopping_list         ,
        : {"    ": 5, "      ": 15, "    ": 7}"""
    def __init__(self, shopping_list):
        self.shopping_list = shopping_list

    """          上          """
    def get_item_count(self):
        return len(self.shopping_list)

    """                          """
    def get_total_price(self):
        total_price = 0
        for price in self.shopping_list.values():
            total_price += price
        return total_price
'''

import unittest
from shopping_list import ShoppingList

class TestShoppingList(unittest.TestCase):
    def setUp(self):
        self.shopping_list = ShoppingList({"    ": 8, "    ": 30, "    ": 15})

    def test_get_item_count(self):
        self.assertEqual(self.shopping_list.get_item_count(), 3)
```

```
    def test_get_total_price(self):
        self.assertEqual(self.shopping_list.get_total_price(), 55)
```