CS50P

> **Tip**
>
> **Python基础：CS50P Introduction to Programming with Python**
>
> 学习时间：2025.03.01 - 2025.03.05

- [x] Introduction
- [x] Functions, Variables
- [x] Conditionals
- [x] Loops
- [x] Exceptions
- [x] Libraries
- [x] Unit Tests
- [x] File I/O
- [x] Regular Expressions
- [x] Object-Oriented Programming
- [x] Et Cetera

Introduction

推荐做法

- **观看讲座**
- **观看短视频**
- **提交编程作业**

Functions, Variables

创建 Python 代码

- 使用 VS Code 进行代码编写和执行。
- `hello.py` 是一个简单的 Python 程序，输出 "hello, world"。
- `print()` 函数用于在终端窗口输出文本。
- `python hello.py` 命令运行 Python 程序。

函数

- 函数是计算机或编程语言已经知道如何执行的动作。
- 函数可以**接受参数**并返回值。
- `print()` 函数是内置函数，用于输出文本。

错误

- 编程中的错误是正常现象，需要解决。
- 错误信息可以帮助您找到错误并修复它。

改进第一个 Python 程序

- 使用 `input()` 函数获取用户输入。

    - ```
      input("What's your name? ")
      print("hello, world")
      ```
    - 实际并未输出用户名，需要变量。
- 使用变量存储用户输入的值。
    - ```
      name = input("What's your name? ")
      print("hello, name")
      ```
    - 使用 `=` 进行变量的赋值操作。
    - 还是没有按照预期输出用户输入的用户名。
- 使用 `print()` 函数输出用户输入的值。
    - ```
      name = input("What's your name? ")
      print("hello,")
      print(name)
      ```
    - 勉强接近预期输出。

变量

- 变量是程序中用于存储值的**容器**。
- 变量需要先声明，然后才能使用。
- 变量可以存储不同类型的值，例如字符串、整数和浮点数。

注释

- 注释是程序员对代码的解释或说明。
- 注释不会影响程序的执行。
- 注释可以提高代码的可读性。
- 同时注释也可以作为to-do list。

伪代码

- 伪代码是用于规划和设计程序的文本表示。
- 伪代码可以帮助更好地理解程序逻辑。
- ```
  # Ask the user for their name
  name = input("What's your name? ")

  # Print hello
  print("hello,")

  # Print the name inputted
  print(name)
  ```

进一步改进第一个 Python 程序

- ```
  # Ask the user for their name
  name = input("What's your name? ")

  # Print hello and the inputted name
  print("hello, " + name)
  ```
- 方法可以接受多个参数
  ```

- 使用`,`进行参数间隔
- ```
  # Ask the user for their name
  name = input("What's your name? ")

  # Print hello and the inputted name
  print("hello,", name)
  ```

字符串和参数

- 字符串是**文本序列**，在Python中表示为`str`。
- 函数可以接受参数，参数可以影响函数的行为。
- `print()`函数可以接受多个参数，参数之间用逗号分隔。
- `print`方法默认包含一个`end='\n'`，所以默认情况下会进行换行输出。
- 可以自己为`end`变量赋值，如下：
  - ```
    # Ask the user for their name
    name = input("What's your name? ")
    print("hello,", end="")
    print(name)
    ```
  - 使用`end=''`覆盖了原有的默认值。

引号的问题

- `print("hello,"friend"")`会报错
- 解决方法：`print("hello, \"friend\"")`，告诉解释器需要将其视作引号。

格式化字符串

- `f-string`是一种格式化字符串的方法。
- `f-string`可以在字符串中插入变量值。
- `f-string`可以格式化数字、字符串和日期等数据类型。
- ```
  # Ask the user for their name
  name = input("What's your name? ")
  print(f"hello, {name}")
  ```

字符串

- 字符串可以包含空格和其他特殊字符。
- `strip()`方法可以删除字符串两端的空格。
- `title()`方法可以将字符串的第一个单词的首字母大写，其他单词的首字母小写。
- ```
  # Ask the user for their name
  name = input("What's your name? ")

  # Remove whitespace from the str
  name = name.strip()

  # Capitalize the first letter of each word
  name = name.title()

  # Print the output
  print(f"hello, {name}")
  ```

- 进一步简化
  - ```
    # Ask the user for their name
    name = input("What's your name? ")

    # Remove whitespace from the str and capitalize the first letter of each word
    name = name.strip().title()

    # Print the output
    print(f"hello, {name}")
    ```
- 更进一步的
  - ```
    # Ask the user for their name, remove whitespace from the str and capitalize
    the first letter of each word
    name = input("What's your name? ").strip().title()

    # Print the output
    print(f"hello, {name}")
    ```

整数

- 整数是没有小数部分的数字，Python中表示为`int`。
- `int()`函数可以将字符串转换为整数。
  - ```
    x = input("What's x? ")
    y = input("What's y? ")

    z = x + y

    print(z)
    ```
  - 输入1和2，但输出结果为12而不是3
  - 注意：键盘上输入会被视为文本，也就是字符串，故而需要将输入转换为`int`
  - ```
    x = input("What's x? ")
    y = input("What's y? ")

    z = int(x) + int(y)

    print(z)
    ```
  - 更进一步的简化：首先会执行`input`，而后会执行`int`
  - ```
    x = int(input("What's x? "))
    y = int(input("What's y? "))

    print(x + y)
    ```
- `+`运算符可以用于整数相加。

可读性

- 编写可读性强的代码非常重要。
- 使用缩进和注释可以提高代码的可读性。

浮点数
    ```

- 浮点数是有小数部分的数字。
- `float()` 函数可以将字符串转换为浮点数。
    - ```
      x = float(input("What's x? "))
      y = float(input("What's y? "))

      print(x + y)
      ```
- `+` 运算符可以用于浮点数相加。
- 浮点数可采用 `round` 进行近似
    - ```
      # Get the user's input
      x = float(input("What's x? "))
      y = float(input("What's y? "))

      # Create a rounded result
      z = round(x + y)

      # Print the result
      print(z)
      ```
    - 是近似到最接近的整数
- 标准化输出长数字，例如 `1000` 输出为 `1,000`
    - ```
      # Get the user's input
      x = float(input("What's x? "))
      y = float(input("What's y? "))

      # Create a rounded result
      z = round(x + y)

      # Print the formatted result
      print(f"{z:,}")
      ```
    - `print(f"{z:,}")`
- 标准化输出?
    - ```
      # Get the user's input
      x = float(input("What's x? "))
      y = float(input("What's y? "))

      # Calculate the result and round
      z = round(x / y, 2)

      # Print the result
      print(z)
      ```
        - 保留两位小数

    - ```
      # Get the user's input
      x = float(input("What's x? "))
      y = float(input("What's y? "))

      # Calculate the result
      z = x / y

      # Print the result
      print(f"{z:.2f}")
      ```

        - 使用了`fstring`来标准化输出。

创建自定义函数

- 使用 `def` 关键字创建自定义函数。
- 函数可以接受参数，并使用 `return` 语句返回值。
- 函数可以提高代码的可重用性。
- 常规写法：
    - ```
      def main():

          # Output using our own function
          name = input("What's your name? ")
          hello(name)

          # Output without passing the expected arguments
          hello()


      # Create our own function
      def hello(to="world"):
          print("hello,", to)


      main()
      ```

    - 需要调用`main`方法才可以使得程序工作。

返回值

- 函数可以返回值，返回值可以用于其他操作。
- ```
  def main():
      x = int(input("What's x? "))
      print("x squared is", square(x))


  def square(n):
      return n * n


  main()
  ```

- `square()` 函数返回平方值。

课外Shorts

- VS Code for CS50
  ```

    - 搭建云端的开发环境
    - 常用的Linux指令：
        - ls、cp、mv、rm、mkdir、cd、rmdir、clear……
- 函数
    - 需要`def`和`calling`
- 变量
    - 存储可变值的容器
    - 键盘获取的输入为字符串，与数字并不相等。
- 返回值
    - 使得函数可以返回可以使用的值。
- 副作用
    - 例如打印到终端
    - 改变一些全局变量
        - ```
          emoticon = "v.v"

          def main():
              global emoticon
              say("Is anyone there?")
              emoticon = ":D"
              say("Oh, hi!")


          def say(phrase):
              print(phrase + " " + emoticon)

          main()
          ```
        - 注意在局部修改全局变量时，Python要求使用`global`关键字，使得其不仅可访问，而且可修改。
- 字符串
    - 字符串方法：这里所指的一类方法为属于某种对象的函数。
    - `.strip()`、`.title()`、`.join()`方法

Problem Set 0

- `indoor.py`
    - 关键点：字符串转为小写
    - ```
      print(input().lower())
      ```
- `playback.py`
    - 关键点：分割字符串并拼接
    - ```
      input = input().split()
      print("...".join(input))
      ```
- `face.py`
    - 关键点：替换字符串中的元素
      ```

    - ```
      def main():
          print(convert(input()))

      def convert(str):
          if ":)" in str:
              str = str.replace(":)", "🙂")
          if ":(" in str:
              str = str.replace(":(", "☹️")
          return str

      main()
      ```

- `einstein.py`
    - 关键点：类型转换
    - ```
      m = int(input("m: "))
      print("E:", m * 300000000 * 300000000)
      ```

- `tip.py`
    - 关键点：`strip`函数处理字符串首尾以及类型转换。
    - ```
      def main():
          dollars = dollars_to_float(input("How much was the meal? "))
          percent = percent_to_float(input("What percentage would you like to tip?
      "))
          tip = dollars * percent
          print(f"Leave ${tip:.2f}")


      def dollars_to_float(d):
          return float(d.strip("$"))


      def percent_to_float(p):
          return float(p.strip("%")) / 100


      main()
      ```

- ❗善于查看文档寻找合适的方法解决问题。

Conditionals

条件语句

- 条件语句允许程序根据特定条件做出决策。
- Python 内置了一系列运算符来比较数值和变量。
- `==` 用于比较两个值是否相等，`!=` 用于比较两个值是否不相等。

if 语句

- `if` 语句根据条件执行代码块。
- 如果条件为真，则执行代码块，否则不执行。
- 例如：
      ```

```
x = int(input("What's x? "))
y = int(input("What's y? "))
if x < y:
    print("x is less than y")
```

控制流、elif 和 else

- 控制流指的是程序执行路径的顺序。
- `elif` 语句用于处理多个条件，如果 `if` 语句的条件为假，则执行第一个 `elif` 语句的条件。
- 使用`elif`语句和`else`而不是一味地使用`if`可以减小判断逻辑。
- `else` 语句用于处理所有其他情况，即所有 `if` 和 `elif` 语句的条件都为假时。

逻辑运算符

- `or` 运算符用于连接多个条件，只要其中一个条件为真，整个条件就为真。同时也要问自己条件是否可以合并，从而更加高效。
- `and` 运算符用于连接多个条件，只有所有条件都为真，整个条件才为真。
- 例如：

```
if x < y or x > y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

模运算符

- `%` 运算符用于计算除法的余数。
- 可以用于判断一个数是奇数还是偶数。
- 例如：

```
x = int(input("What's x? "))
if x % 2 == 0:
    print("Even")
else:
    print("Odd")
```

创建自定义函数

- 使用 `def` 关键字创建自定义函数。
- 函数可以接受参数，并使用 `return` 语句返回值。
- 例如：

```
def main():
    x = int(input("What's x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
main()
```

Pythonic 编码

- Pythonic 编码是指使用 Python 风格编写代码。
- 通常更简洁、更易读。
- 例如：

```
def is_even(n):
    return n % 2 == 0
```

match 语句

- `match` 语句用于根据值执行不同的代码块。
- 可以用于替代多个 `if` 和 `elif` 语句。
- 例如：

```
name = input("What's your name? ")
match name:
    case "Harry":
        print("Gryffindor")
    case "Hermione":
        print("Gryffindor")
    case "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

- 使用 `_` 匹配任何输入，相当于 `else`
- 注意只会成果匹配一次，其后中止匹配。
- 进一步简化上述代码：
- ```
  name = input("What's your name? ")

  match name:
    case "Harry" | "Hermione" | "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
  ```
  ```

课外Shorts

- 条件：提出一个有是否的问题，根据条件选择不同的路径。
  - 不要过于信任用户?
  - ```
    def main():
        difficulty = input("Difficult or Casual? ")
        players = input("Multiplayer or Single-player? ")

        if difficulty == "Difficult":
            if players == "Multiplayer":
                recommend("Poker")
            elif players == "Single-player":
                recommend("Klondike")
            else:
                print("Enter a valid number of players")
        elif difficulty == "Casual":
            if players == "Multiplayer":
                recommend("Hearts")
            elif players == "Single-player":
                recommend("Clock")
            else:
                print("Enter a valid number of players")
        else:
            print("Enter a valid difficulty")


    def recommend(game):
        print("You might like", game)


    main()
    ```

- 布尔表达式
  - `not`进行取反操作
  - 利用`and`或`or`进行逻辑的扩展。
  - ```
    def main():
        difficulty = input("Difficult or Casual? ")
        if not (difficulty == "Difficult" or difficulty == "Casual"):
            print("Enter a valid difficulty")
            return

        players = input("Multiplayer or Single-player? ")
        if not (players == "Multiplayer" or players == "Single-player"):
           print("Enter a valid number of players")
           return

        if difficulty == "Difficult" and players == "Multiplayer":
            recommend("Poker")
        elif difficulty == "Difficult" and players == "Single-player":
            recommend("Klondike")
        elif difficulty == "Casual" and players == "Multiplayer":
            recommend("Hearts")
        else:
            recommend("Clock")
    ```

```
def recommend(game):
    print("You might like", game)


main()
```

Problem Set 1

- `deep.py`
  - 关键点：统一成小写和去掉前后空格。
  - ```
    answer = input("What is the Answer to the Great Question of Life, the Universe, and Everything? ").lower().strip()
    if answer == "42" or answer == "forty-two" or answer == "forty two":
        print("Yes")
    else:
        print("No")
    ```
- `bank.py`
  - 关键点：使用`startswith`判断是否以某字符串开头。
  - ```
    greeting = input("Greeting: ").lower().strip()

    if greeting.startswith("hello"):
        print("$0")
    else:
        if greeting.startswith("h"):
            print("$20")
        else:
            print("$100")
    ```
- `extensions.py`
  - 关键点：字符串分割`split`，条件较多时可以使用`match`
  - ```
    file = input("File name: ").split(".")

    if len(file) == 1:
        print("application/octet-stream")
    else:
        extension = file[-1].lower().strip()
        match extension:
            case "gif":
                print("image/gif")
            case "jpg" | "jpeg":
                print("image/jpeg")
            case "png":
                print("image/png")
            case "pdf":
                print("application/pdf")
            case "txt":
                print("text/plain")
            case "zip":
                print("application/zip")
            case _:
                print("application/octet-stream")
    ```
    ```

- `interpreter.py`
  - 关键点：浮点数输出标准化，`split`分割输入。
  - ```
    expression = input("Expression: ")
    x, y, z = expression.split(" ")
    x = float(x)
    z = float(z)

    match y:
        case '+':
            print(f"{x + z:.1f}")
        case '-':
            print(f"{x - z:.1f}")
        case '*':
            print(f"{x * z:.1f}")
        case '/':
            print(f"{x / z:.1f}")
    ```
- `meal.py`
  - 关键点：可以采用多个运算符进行逻辑运算
  - ```
    def main():
        time = input("What time is it? ")
        convert_time = convert(time)
        if 7.0 <= convert_time <= 8.0:
            print("breakfast time")
        elif 12.0 <= convert_time <= 13.0:
            print("lunch time")
        elif 18.0 <= convert_time <= 19.0:
            print("dinner time")


    def convert(time):
        hours, minutes = time.split(":")
        hours = int(hours)
        minutes = int(minutes)
        return hours + minutes / 60


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

Loops

循环

- 循环使得可以一遍又一遍地重复执行代码块。

While 循环

- `while` 循环可以重复执行一个代码块，**直到指定的条件不再满足。**
- 示例代码：

```
i = 3
while i != 0:
    print("meow")
    i -= 1
```

- 注意：避免创建无限循环，确保循环条件最终会变为假。
- 陷入死循环时可以采用`ctrl + c`打断循环。
- 循环的**最佳实践是从0开始**计数
- ```
  i = 0
  while i < 3:
      print("meow")
      i += 1
  ```

For 循环

- `for` 循环用于遍历序列（如列表、元组、字典、集合或字符串）。

```
for i in [0, 1, 2]:
    print("meow")
```

- 应对极端情况，使用`range`，示例代码：

```
for i in range(3):
    print("meow")
```

- 使用 `_` 作为循环变量，当变量在循环体中不被使用时。
- 进一步改进代码，Python代码的可能性
- ```
  print("meow\n" * 3, end="")
  ```

改进用户输入

- 使用 `while` 循环来验证用户输入。
- 示例代码：

```
while True:
    n = int(input("What's n? "))
    if n > 0:
        break
```

- `continue` 和 `break` 关键字用于控制循环流程。

列表`list`

- 列表是一种有序的集合，可以包含不同类型的元素。
- 示例代码：

```
students = ["Hermione", "Harry", "Ron"]
for student in students:
    print(student)
```

- `len()` 函数用于获取列表的长度。

字典`dict`

- 字典是一种键值对的数据结构。
- `list`使用下表遍历，而`dict`可以使用键

    - ```
      students = {
          "Hermione": "Gryffindor",
          "Harry": "Gryffindor",
          "Ron": "Gryffindor",
          "Draco": "Slytherin",
      }
      print(students["Hermione"])
      print(students["Harry"])
      print(students["Ron"])
      print(students["Draco"])
      ```
    - 改进
    - ```
      students = {
          "Hermione": "Gryffindor",
          "Harry": "Gryffindor",
          "Ron": "Gryffindor",
          "Draco": "Slytherin",
      }
      for student in students:
          print(student)
      ```
        - 注意：上述代码只会遍历字典的键而不会遍历值。
- 示例代码：

```
students = {
    "Hermione": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin",
}
for student in students:
    print(student, students[student], sep=", ")
```

- 字典使用 `{}` 创建，键和值之间用 `:` 分隔。
- ```
  students = [
      {"name": "Hermione", "house": "Gryffindor", "patronus": "Otter"},
      {"name": "Harry", "house": "Gryffindor", "patronus": "Stag"},
      {"name": "Ron", "house": "Gryffindor", "patronus": "Jack Russell terrier"},
      {"name": "Draco", "house": "Slytherin", "patronus": None},
  ]

  for student in students:
      print(student["name"], student["house"], student["patronus"], sep=", ")
  ```
    - 列表加字典存储大量关联数据

Mario 游戏文本表示

- 使用循环来创建 Mario 游戏中的砖块表示。
- 示例代码：
  ```

```
def main():
    print_square(3)
def print_square(size):
    for i in range(size):
        print_row(size)
def print_row(width):
    print("#" * width)
main()
```

- 通过嵌套循环创建行和列。

总结

- 学习了 `while` 和 `for` 循环的使用。
- 掌握了列表和字典的基本操作。
- 学会了如何使用循环来处理用户输入和创建简单的文本图形。

课外Shorts

- 字典
  - 需根据**键值对**存储类似数据时十分有用。
  - 可以组合信息并使用键很容易的取出值。
  - 不仅可以在初始化时指定有哪些键，可以后续直接添加即可。
  - 不仅可以使用中括号获取键值，也可以使用 `get` 方法，可避免访问不存在的键。
  - 可以使用 `update` 方法传入另一本字典实现多个键值扩充。
  - `.keys()` 方法可返回字典中所有键。
  - `.values()` 方法可返回字典中所有值。
- 字典方法
  - `len` 方法获取字典长度。
  - `pop(KEY)` 弹出字典中该键值对。
  - `clear()` 方法清空字典。
  - 迭代：`for key, value in Dic.items()`
- `for` 循环
  - 重复的事情。
  - 对于列表的迭代。
- 字典和列表的表达式
  - `[word.lower() for word in words if len(word) > 4]`
  - 上述写法可以迭代并对原数据进行处理。
  - ```
    def main():
        counts = {}
        words = get_words("address.txt")
        words = [word.lower() for word in words if len(word) > 4]

        counts = {word: words.count(word) for word in words}

        save_counts(counts)
    ```
  - `list` 和 `dic` 都可以采用上述这种更为接近于口语的表达简化代码。
    ```

- `list`方法
  - `append`方法增加
  - `pop`弹出最后的元素。
  - `clear`清空所有元素。
- 字符串切片
  - `phone[0:3]`，左闭右开。
    - 左边数字或者右边数字可以省略
  - 从后往前？`phone[-4:]`，转一圈最后的字符为`-1`
- 元组`Tuples`
  - 为什么不用`list`？
    - 元组不支持重新复制操作。
    - 只可以在初始化时赋值。
    - 同时元组相较于列表更加有效，节省内存。
- `While`循环
  - 条件为真时就会一直执行循环语句。

Problem Set 2

- `camel.py`
  - 关键点：字符串的迭代器遍历。
  - ```
    camel_case = input("camelCase: ")

    print("snake_case: ", end="")

    for ch in camel_case:
        if ch.isupper():
            print(f"_{ch.lower()}", end="")
        else:
            print(ch, end="")

    print()
    ```
- `coke.py`
  - 关键点：循环语句的使用。
  - ```
    amount = 50

    while amount > 0:
        print(f"Amount Due: {amount}")
        coin = int(input("Insert Coin: "))
        if coin == 25 or coin == 10 or coin == 5:
            amount -= coin

    print(f"Change Owed: {-amount}")
    ```
- `tettr.py`
  - 关键点：结合前文的`in`可以简化条件判断。

```
text = input("Input: ")

print("Output: ", end="")

for c in text:
    if c.upper() not in ['A', 'E', 'I', 'O', 'U']:
        print(c, end="")
print()
```

- `plates.py`
  - 关键点：依据条件进行判断以及字符串分片。
  - ```
    def main():
        plate = input("Plate: ")
        if is_valid(plate):
            print("Valid")
        else:
            print("Invalid")


    def is_valid(s):
        index = 0
        if not (2 <= len(s) <= 6):
            return False
        for c in s:
            if not ('A' <= c <= 'Z' or '0' <= c <= '9'):
                return False
            else:
                if 'A' <= c <= 'Z':
                    index += 1
        if not ('A' <= s[0] <= 'Z' and 'A' <= s[1] <= 'Z'):
            return False

        if (s[index:].isdigit() and s[index] != '0') or index == len(s):
            return True

        return False


    main()
    ```

- `nutrition.py`
  - 关键点：使用字典存储键值简化判断。
  - ```
    fruit = {
        "Apple": 130,
        "Avocado": 50,
        "Banana": 110,
        "Cantaloupe": 50,
        "Grapefruit": 60,
        "Grapes": 90,
        "Honeydew Melon": 50,
        "Kiwifruit": 90,
        "Lemon": 15,
        "Lime": 20,
        "Nectarine": 60,
        "Orange": 80,
        "Peach": 60,
    ```

```
    "Pear": 100,
    "Pineapple": 50,
    "Plums": 70,
    "Strawberries": 50,
    "Sweet Cherries": 100,
    "Tangerine": 50,
    "Watermelon": 80
}

item = input("Item: ")

if item.title() in fruit:
    print("Calories:", fruit[item.title()])
```

Exceptions

异常（Exceptions）

- 异常是代码中出错的情况。
- 在Python中，异常处理是一种重要的编程实践，可以帮助我们更好地管理程序中的错误。

创建异常

- 在文本编辑器中创建一个名为 `hello.py` 的文件，输入以下代码（包含故意设置的错误）：

```
print("hello, world)
```

- 注意，我们故意遗漏了一个引号。
- 在终端窗口中运行 `python hello.py`，会输出一个错误，编译器会指出这是一个“语法错误”。

运行时错误（Runtime Errors）

- 运行时错误是由代码中的意外行为引起的。
- 例如，用户可能被要求输入一个数字，但他们却输入了一个字符。这种意外的用户输入可能会导致程序抛出错误。

```
x = int(input("What's x? "))
print(f"x is {x}")
```

- 如果用户输入“cat”而不是一个数字，程序会抛出 `ValueError`。

try 语句

- `try` 和 `except` 语句用于在出错前测试用户输入。
- 修改代码如下：

```
try:
    x = int(input("What's x?"))
    print(f"x is {x}")
except ValueError:
    print("x is not an integer")
```

- 如果用户输入正确，将接受并打印。
- 如果输入错误，则会捕获异常并给出提示。

else 语句

- 如果 `try` 块中没有异常发生，可以使用 `else` 语句来执行代码。
- 修改代码如下：

```
try:
    x = int(input("What's x?"))
except ValueError:
    print("x is not an integer")
else:
    print(f"x is {x}")
```

创建获取整数的函数

- 我们可以将获取整数的逻辑抽象成一个函数。

```
def main():
    x = get_int()
    print(f"x is {x}")
def get_int():
    while True:
        try:
            x = int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
        else:
            return x
main()
```

- 进一步优化，使用 `pass` 语句可以让代码在用户输入错误时不给出警告，而是重新询问。

```
def get_int(prompt):
    while True:
        try:
            return int(input(prompt))
        except ValueError:
            pass
```

总结

- 错误在代码中是不可避免的，但我们可以使用今天学到的知识来预防这些错误。
- 在本节课中，我们学习了以下内容：
    - 异常（Exceptions）
    - 值错误（Value Errors）
    - 运行时错误（Runtime Errors）
    - try 语句
    - else 语句
    - pass 语句

课外Shorts

- `Debugging`
    - **打印**出函数中的变量看是否符合预期。
    - 更好的工具?
        - 断点，将程序暂停下来。

        - DEBUG工具
- 处理异常
    - 尽量详细地写出异常的类型。
- 创建异常
    - 不仅仅是打印错误提醒用户。
    - `raise`关键字，同样的错误需要更具体一些。

Problem Set 3

- `fuel.py`
    - ```
      fraction = input("Fraction: ")

      try:
          x, y = fraction.split('/')
          x = int(x)
          y = int(y)
          if x > y:
              raise ValueError
          z = round(x / y * 100)
      except (ValueError, ZeroDivisionError):
          fraction = input("Fraction: ")
          x, y = fraction.split('/')
          x = int(x)
          y = int(y)
          z = round(x / y * 100)

      if z <= 1:
          print("E")
      elif z >= 99:
          print("F")
      else:
          print(f"{z}%")
      ```
    - 关键点：`round`用以近似，对于错误的捕捉抛出。
- `taqueria.py`
    - ```
      menu = {
          "Baja Taco": 4.25,
          "Burrito": 7.50,
          "Bowl": 8.50,
          "Nachos": 11.00,
          "Quesadilla": 8.50,
          "Super Burrito": 8.50,
          "Super Quesadilla": 9.50,
          "Taco": 3.00,
          "Tortilla Salad": 8.00
      }

      total = 0.00

      while True:
          try:
              item = input("Item: ")
              if item.title() in menu:
      ```

```
            total += menu[item.title()]
            print(f"Total: ${total:.2f}")
    except EOFError:
        break
```

- 关键点：捕获 `ctrl+d`，判断字典中是否存在对应 `key`

- `grocery.py`

```
items = {}

while True:
    try:
        item = input().upper()
        if item in items:
            items[item] += 1
        else:
            items[item] = 1
    except EOFError:
        names = sorted(items)
        for name in names:
            print(items[name], name)
        break
```

- 关键点：对于是否在原字典中的处理。`sorted` 处理后返回的是 `list`。

- `outdated.py`

```
months = [
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
]

while True:
    try:
        date = input("Date: ")
        date_list = date.split('/')
        if len(date_list) == 3:
            year = int(date_list[2])
            month = int(date_list[0])
            day = int(date_list[1])
        else:
            date_list = date.split()
            year = int(date_list[2])
            for i in range(12):
                if months[i] == date_list[0]:
                    month = i + 1
```

```
            day = int(date_list[1][:-1])
        if day > 31 or month > 12:
            raise ValueError
    except (ValueError, NameError):
        continue
    else:
        print(f"{year:04}-{month:02}-{day:02}")
        break
```

- 关键点：代码写的过于臃肿了，关键在于字符串的处理提取出有用信息并转换。

Libraries

库 (Libraries)

- 库是可重用的代码块，可以导入到程序中以提高生产力。
- Python 允许你将函数或功能作为模块共享。
- 你可以从旧项目中复制和粘贴代码来创建模块或库。

随机库 (Random)

- `random` 是一个内置库，用于生成随机数和随机选择。
- 使用 `import random` 来导入库。
- 使用 `random.choice(seq)` 从序列中选择一个随机项。
  - ```
    import random

    coin = random.choice(["heads", "tails"])
    print(coin)
    ```
  - `import` 了 `random` 的全部内容，如何改进?
  - ```
    from random import choice

    coin = choice(["heads", "tails"])
    print(coin)
    ```
  - 明确只需要一部分内容时，可采取上述方法。
- 使用 `random.randint(a, b)` 生成指定范围内的随机整数。
  - ```
    import random

    number = random.randint(1, 10)
    print(number)
    ```
- 使用 `random.shuffle(x)` 将列表随机排序。
  - ```
    import random

    cards = ["jack", "queen", "king"]
    random.shuffle(cards)
    for card in cards:
        print(card)
    ```
  - 注意该方法没有返回值，是针对原有 `list` 中内容的随机排序。

统计库 (Statistics)

- `statistics` 是一个内置库，提供统计功能。
- 使用 `import statistics` 来导入库。
- 使用 `statistics.mean()` 计算平均值。

命令行参数 (Command-Line Arguments)

- `sys` 库允许从命令行获取参数。
- 使用 `sys.argv` 来访问命令行参数列表。
  - 示例：`python name.py David`，`sys.argv[0]` 对应 `name.py`，`argv[1]` 对应 `David`

```
import sys

print("hello, my name is", sys.argv[1])
```

  - 健壮性
  - ```
    import sys

    try:
        print("hello, my name is", sys.argv[1])
    except IndexError:
        print("Too few arguments")
    ```
  - 进一步改进。
  - ```
    import sys

    if len(sys.argv) < 2:
        print("Too few arguments")
    elif len(sys.argv) > 2:
        print("Too many arguments")
    else:
        print("hello, my name is", sys.argv[1])
    ```
  - 给予用户提示。
- 使用 `sys.exit()` 来终止程序。
  - ```
    import sys

    if len(sys.argv) < 2:
        sys.exit("Too few arguments")
    elif len(sys.argv) > 2:
        sys.exit("Too many arguments")

    print("hello, my name is", sys.argv[1])
    ```

切片 (Slice)

- 切片操作符 `[:]` 可以用来获取列表的一部分。
- 使用 `sys.argv[1:]` 可以获取除了脚本名称之外的所有命令行参数。
    ```

- ```python
  import sys

  if len(sys.argv) < 2:
      sys.exit("Too few arguments")

  for arg in sys.argv[1:]:
      print("hello, my name is", arg)
  ```

包 (Packages)

- 包是第三方库，提供额外的功能。
- PyPI 是一个包含所有可用第三方包的**仓库**。
- 使用 `pip` 工具可以安装包。
- 例如，使用 `pip install cowsay` 安装 cowsay 包。

API

- API 允许你连接到其他代码。
- `requests` 库允许你的程序发送 HTTP 请求。可使用 `.json` 获取响应的内容。
- `json` 库可以帮助你解析和生成 JSON 数据。
- 例如，使用 `requests.get()` 发送 HTTP GET 请求。
- 使用 `json.dumps()` 将 JSON 数据格式化输出。

创建自己的库

- 你可以创建自己的库，以便重用代码或与他人共享。
- 使用 `from module import function` 可以导入特定函数。

课外Shorts

- **API调用**
  - 可采用API在代码其他地方调用或在互联网上使用。
  - API可以包含参数，注意查阅API文档。
- **创建模块和包**
  - 使用模块可以将功能抽象出来，使得其他地方也可以使用。
  - 包就是一个文件夹，可以在其中新建一个 `__init__.py` 文件提醒这是一个多模块的包。将自己的模块整合到一个包下是共享代码的一个好方法。
- **random**
  - `choice` 方法随机选择一个。
  - `choices` 方法需要参数 `k` 选择多个，但是是有放回的。
  - `sample` 方法是无放回的。
  - 可以传入权重参数使得有所侧重。例 `weights=[100, 0, 0]`
  - 随机难以调试，但可以使用 `seed` 将其明确。
- **风格**
  - 相对较少的代码保持功能和可读性。
  - Python存在一些严格的遵守PEP 8。
  - 可读性很重要，有关一致性。
  - **缩进**，四个空格，最大长度限制、添加空行、导入的位置。
  ```

- `pylint`工具可以帮助我们。`black`工具也正在兴起。

Problem Set 4

- `emojize.py`
  - ```
    from emoji import emojize

    def main():
        emoji = input("Input: ")
        print(f"Output: {emojize(emoji, language='alias')}")

    main()
    ```
  - 关键点：调用`emoji`库
- `figlet.py`
  - ```
    from pyfiglet import Figlet
    import sys
    import random

    def main():
        figlet = Figlet()

        if len(sys.argv) == 3:
            if sys.argv[1] in ["-f", "--font"]:
                if sys.argv[2] in figlet.getFonts():
                    figlet.setFont(font=sys.argv[2])
                else:
                    sys.exit("Invalid usage")
            else:
                sys.exit("Invalid usage")
        elif len(sys.argv) == 1:
            figlet.setFont(font=random.choice(figlet.getFonts()))
        else:
            sys.exit("Invalid usage")
        s = input("Input: ")
        print(figlet.renderText(s))

    main()
    ```
  - 关键点：`sys`获取命令行参数，其他库函数的使用。
- `adieu.py`
  - ```
    import inflect

    p = inflect.engine()

    names = []

    while True:
        try:
            name = input("Name: ")
            names.append(name)
        except EOFError:
            print(f"Adieu, adieu, to {p.join(names)}")
            break
    ```
    ```

- 关键点：查看`inflect`库函数的使用。

- `game.py`

```
import random
import sys

def main():
    level = get_level()
    number = random.randint(1, level)
    while True:
        try:
            guess = int(input("Guess: "))
        except ValueError:
            continue
        if guess <= 0:
            continue
        if guess == number:
            print("Just right!")
            sys.exit()
        elif guess < number:
            print("Too small!")
        else:
            print("Too large!")

def get_level():
    while True:
        try:
            level = int(input("Level: "))
            if level <= 0:
                raise ValueError
        except ValueError:
            pass
        else:
            return level

main()
```

- 关键点：`random`库函数使用以及`sys`快速中止程序。

- `professor.py`

```
import random


def main():
    level = get_level()
    count = 0
    score = 0
    while count < 10:
        x = generate_integer(level)
        y = generate_integer(level)
        cnt = 0
        while cnt < 3:
            try:
                result = int(input(f"{x} + {y} = "))
            except ValueError:
                cnt += 1
                print("EEE")
            else:
```

```
                if result == x + y:
                    count += 1
                    score += 1
                    break
                else:
                    cnt += 1
                    print("EEE")
        if cnt == 3:
            count += 1
            print(f"{x} + {y} = {x + y}")

    print(f"Score: {score}")



def get_level():
    while True:
        try:
            level = int(input("Level: "))
        except ValueError:
            continue
        else:
            if level in [1, 2, 3]:
                return level


def generate_integer(level):
    if level == 1:
        return random.randint(0, 9)
    elif level == 2:
        return random.randint(10, 99)
    else:
        return random.randint(100, 999)


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

  - 关键点：循环条件的判断。

- `bitcoin.py`

  ```
  import sys
  import requests


  def main():
      if len(sys.argv) == 1:
          sys.exit("Missing command-line argument")
      elif len(sys.argv) == 2:
          try:
              number = float(sys.argv[1])
          except ValueError:
              sys.exit("Command-line argument is not a number")
          else:
              try:
                  response =
  requests.get("https://api.coincap.io/v2/assets/bitcoin")
                  output = response.json()
                  print(f"${float(output["data"]["priceUsd"]) * number:,.4f}")
  ```

```
            except requests.RequestException:
                pass

main()
```

- 关键点：获取命令行参数以及`requests`库的使用，格式化也需关注。

Unit Tests

单元测试 (Unit Tests)

- 单元测试是编程过程中的自然部分，用于测试代码的特定方面。
- 你可以创建自己的测试程序来测试你的代码。
  - ```
    from calculator import square


    def main():
        test_square()


    def test_square():
        if square(2) != 4:
            print("2 squared was not 4")
        if square(3) != 9:
            print("3 squared was not 9")


    if __name__ == "__main__":
        main()
    ```
  - 上面的代码就是一个简单的测试代码，测试一些条件下程序是否符合预期。
- 或者，你可以使用像`pytest`这样的框架来运行你的单元测试。

`assert`语句

- `assert`语句允许你告诉编译器某个断言为真。
- 它用于测试代码中的条件。
  - ```
    from calculator import square

    def main():
        test_square()

    def test_square():
        assert square(2) == 4
        assert square(3) == 9


    if __name__ == "__main__":
        main()
    ```
  - 就像上面所展示的那样，可以简化判断。
- 如果断言失败，则会抛出`AssertionError`。

`pytest`

- `pytest` 是一个第三方库，允许你单元测试你的程序。
- 使用 `pip install pytest` 安装 `pytest`。
- `pytest` 允许你直接运行你的程序，以便更容易地查看测试条件的结果。
- 使用 `pytest test_calculator.py` 运行测试。
- `pytest` 会运行每个测试函数，即使其中一个失败了也会继续运行。
    - 将测试进行分组。
- 使用 `pytest.raises(TypeError)` 来测试期望抛出的错误类型。
    - ```
      import pytest

      from calculator import square

      def test_positive():
          assert square(2) == 4
          assert square(3) == 9


      def test_negative():
          assert square(-2) == 4
          assert square(-3) == 9

      def test_zero():
          assert square(0) == 0

      def test_str():
          with pytest.raises(TypeError):
              square("cat")
      ```

测试字符串

- 测试打印字符串的函数需要修改函数以**返回**字符串，而不是直接打印。
- 使用 `assert` 语句来测试函数返回的字符串值。

组织测试到文件夹

- 单元测试通常需要多个测试，你可以使用 `pytest` 来运行整个测试文件夹。
- 使用 `mkdir test` 创建一个名为 `test` 的文件夹。
- 使用 `code test/test_hello.py` 在 `test` 文件夹中创建测试文件。
- 使用 `code test/__init__.py` 创建一个空的 `__init__.py` 文件，以便 `pytest` 能够识别测试文件夹。
- 使用 `pytest test` 运行整个测试文件夹。

总结

- 测试你的代码是编程过程中的一个自然部分。
- 单元测试允许你测试代码的特定方面。
- 你可以创建自己的测试程序，或者使用像 `pytest` 这样的框架来运行你的单元测试。

课外Shorts

- Pytest
    - 依照惯例测试文件名以 `test_` 开头。
    - 会自动执行这些文件中的函数，函数也需确保以 `test_` 开头。
      ```

- 浮点数很特殊，存在精确程度。可使用`pytest.approx`方法存在容错。

Problem Set 5

- 改进`twttr.py`
  - ```
    def main():

        text = input("Input: ")
        output = shorten(text)

        print("Output: ", output)


    def shorten(word):
        output = ""
        for c in word:
            if c.upper() not in ['A', 'E', 'I', 'O', 'U']:
                output += c
        return output


    if __name__ == "__main__":
        main()
    ```
  - `test_twttr.py`
  - ```
    from twttr import shorten

    def test_shorten():
        assert shorten("What's your name?") == "Wht's yr nm?"
        assert shorten("CS50") == "CS50"
        assert shorten("PYTHON") == "PYTHN"
    ```
  - 编写测试用例并使用`pytest`测试。
- 改进`bank.py`
  - ```
    def main():
        greeting = input("Greeting: ").lower().strip()
        v = value(greeting)
        print(f"${v}")

    def value(greeting):
        if greeting.startswith("hello"):
            return 0
        else:
            if greeting.startswith("h"):
                return 20
            else:
                return 100

    if __name__ == "__main__":
        main()
    ```
  - `test_bank.py`
  - ```
    from bank import value
    ```

```
import pytest

def test_0_value():
    assert value("hello") == 0

def test_20_value():
    assert value("how you doing?") == 20

def test_100_value():
    assert value("what's up?") == 100

def test_case():
    assert value("HELLO") == 0
    assert value("How you doing?") == 20
```

- `test_plates.py`
  - ```
    import plates

    def test_valid():
        assert plates.is_valid("CS50") == True
        assert plates.is_valid("ECTO88") == True

    def test_invalid_01():
        assert plates.is_valid("CS05") == False

    def test_invalid_02():
        assert plates.is_valid("50") == False

    def test_invalid_03():
        assert plates.is_valid("H") == False

    def test_invalid_04():
        assert plates.is_valid("CS50P2") == False

    def test_invalid_05():
        assert plates.is_valid("OUTATIME") == False

    def test_invalid_06():
        assert plates.is_valid("PI3.14") == False
    ```
  - 关键点：测试用例的覆盖范围。
- `test_fuel.py`
  - ```
    from fuel import convert, gauge
    import pytest

    def test_convert():
        assert convert("3/4") == 75

    def test_gauge():
        assert gauge(75) == "75%"

    def test_value_error():
        with pytest.raises(ValueError):
            convert("4/3")
    ```

```
def test_zero_error():
    with pytest.raises(ZeroDivisionError):
        convert("4/0")

def test_E():
    assert gauge(1) == "E"

def test_F():
    assert gauge(99) == "F"
```

File I/O

文件 I/O (File I/O)

- 文件 I/O 允许程序读写文件，将信息存储在文件中**以便以后使用。**
- 使用 `open` 函数打开文件，指定文件名和模式（如 "r" 读取，"w" 写入，"a" 追加）。
- 使用 `with` 语句自动管理文件的打开和关闭，确保文件在使用后正确关闭。

`open` 函数

- `open` 函数用于打开文件，并返回一个文件对象。
- 使用 `file.write()` 方法将数据写入文件。
- 使用 `file.close()` 方法关闭文件。
- ```
  name = input("What's your name? ")

  file = open("names.txt", "w")
  file.write(name)
  file.close()
  ```
  - 上述代码每次会覆盖写。
- ```
  name = input("What's your name? ")

  file = open("names.txt", "a")
  file.write(name)
  file.close()
  ```
  - 模式为`a`表示追加写。
  - ```
    name = input("What's your name? ")

    file = open("names.txt", "a")
    file.write(f"{name}\n")
    file.close()
    ```
  - 改进：增加间隙和换行。
- 采取这种方法读写文件容易忘记关闭文件。

`with` 语句

- `with` 语句用于**自动**管理文件的打开和关闭。
- 使用 `with open("file.txt", "w") as file:` 可以自动关闭文件。
    ```

- ```
  name = input("What's your name? ")

  with open("names.txt", "a") as file:
      file.write(f"{name}\n")
  ```

- 读取文件
  - ```
    with open("names.txt", "r") as file:
        lines = file.readlines()

    for line in lines:
        print("hello,", line)
    ```

    - 采用`r`模式，`readlines()`方法可读取多行并将其存储在`list`中。
    - 解决多换行问题。
      - ```
        with open("names.txt", "r") as file:
            lines = file.readlines()

        for line in lines:
            print("hello,", line.rstrip())
        ```

      - 采用`rstrip()`方法即可。
    - 更进一步简化代码
      - ```
        with open("names.txt", "r") as file:
            for line in file:
                print("hello,", line.rstrip())
        ```

CSV 文件

- CSV (Comma-Separated Values) 文件是一种以逗号分隔值的文本文件格式。
  - 读取CSV文件
    - ```
      with open("students.csv") as file:
          for line in file:
              row = line.rstrip().split(",")
              print(f"{row[0]} is in {row[1]}")
      ```

  - 简化上述代码
    - ```
      with open("students.csv") as file:
          for line in file:
              name, house = line.rstrip().split(",")
              print(f"{name} is in {house}")
      ```

    - 直接将值分配到两个变量中。

    - ```
      students = []

      with open("students.csv") as file:
          for line in file:
              name, house = line.rstrip().split(",")
              students.append(f"{name} is in {house}")

      for student in sorted(students):
          print(student)
      ```

        - `list`中存储字符串，并进行排列。
    - ```
      students = []

      with open("students.csv") as file:
          for line in file:
              name, house = line.rstrip().split(",")
              students.append({"name": name, "house": house})


      def get_name(student):
          return student["name"]


      for student in sorted(students, key=get_name):
          print(f"{student['name']} is in {student['house']}")
      ```

        - `list`中也可以存储多组`dict`
        - 由于字典无法直接排序，需指定**如何获取**需要按照排序的关键字。
        - 可采用`lambda`表达式的方式简化。
            - ```
              students = []

              with open("students.csv") as file:
                  for line in file:
                      name, house = line.rstrip().split(",")
                      students.append({"name": name, "house": house})

              for student in sorted(students, key=lambda student: student["name"]):
                  print(f"{student['name']} is in {student['house']}")
              ```
- 使用`csv`库避免一些错误。
- 使用`csv.reader(file)`可以读取 CSV 文件，返回一个迭代器，每次迭代返回一行数据。
    - ```
      import csv

      students = []

      with open("students.csv") as file:
          reader = csv.reader(file)
          for row in reader:
              students.append({"name": row[0], "home": row[1]})

      for student in sorted(students, key=lambda student: student["name"]):
          print(f"{student['name']} is from {student['home']}")
      ```

- `csv.DictReader`适合于存在`csv`头部的时候
  - ```
    import csv

    students = []

    with open("students.csv") as file:
        reader = csv.DictReader(file)
        for row in reader:
            students.append({"name": row["name"], "home": row["home"]})

    for student in sorted(students, key=lambda student: student["name"]):
        print(f"{student['name']} is in {student['home']}")
    ```
  - 每次读出的是一个字典。
- 使用 `csv.writer(file, fieldnames)` 可以写入 CSV 文件，`fieldnames` 指定列名。
  - ```
    import csv

    name = input("What's your name? ")
    home = input("Where's your home? ")

    with open("students.csv", "a") as file:
        writer = csv.DictWriter(file, fieldnames=["name", "home"])
        writer.writerow({"name": name, "home": home})
    ```
  - `writerow`方法以一个字典作为参数，告知每行写入两个字段。

PIL (Pillow)

- PIL (Python Imaging Library) 是一个用于处理图像的库。
- 使用 `Image.open(filename)` 打开图像文件。
- 使用 `image.save("output.gif", save_all=True, append_images=[image], duration=200, loop=0)` 保存图像为 GIF 动画。

总结

- 文件 I/O 允许程序读写文件，将信息存储在文件中以便以后使用。
- 使用 `open` 和 `with` 语句可以打开和关闭文件。
- 使用 CSV 库可以处理 CSV 文件。
- 使用 PIL 库可以处理图像文件。

课外Shorts

- `Pillow`
  - Python中处理图像的库。
  - `import PIL`
  - `from PIL import Image`，导入库中的特定类。
  - 使用`with`自动关闭。
  - 旋转、保存、过滤

    - ```
      from PIL import Image
      from PIL import ImageFilter


      def main():
          with Image.open("in.jpeg") as img:
              img = img.rotate(180)
              img = img.filter(ImageFilter.FIND_EDGES)
              img.save("out.jpeg")


      main()
      ```

- `Reading and Writing CSVs`
    - CSV适合存储逗号分割的内容，适合数据处理。
    - 可以采用逗号形式打开多个文件。
    - `writer.writeheader()` 写标题。
    - 由 `DictReader` 每次读出的是字典，故可直接对字典进行操作，例如新增字段。
- `Reading and Writing Files`
    - 指定打开模式。
    - `readlines()` 将每行内容读取成字符串列表，`writelines()` 写入字符串列表到每一行。

Problem Set 6

- `lines.py`
    - ```
      import sys

      def main():
          if len(sys.argv) == 1:
              sys.exit("Too few command-line arguments")
          elif len(sys.argv) > 2:
              sys.exit("Too many command-line arguments")

          if not sys.argv[1].endswith(".py"):
              sys.exit("Not a Python file")

          try:
              file = open(sys.argv[1], "r")
          except FileNotFoundError:
              sys.exit("File does not exist")

          count = 0
          for line in file:
              if not (line.lstrip().startswith("#") or line.lstrip() == ""):
                  count += 1

          file.close()
          print(count)


      main()
      ```
    - 关键点：文件的读取，对于命令行参数的获取处理。
- `pizza.py`

    - ```
      import sys
      import csv
      from tabulate import tabulate

      def main():
          if len(sys.argv) == 1:
              sys.exit("Too few command-line arguments")
          elif len(sys.argv) > 2:
              sys.exit("Too many command-line arguments")

          if not sys.argv[1].endswith(".csv"):
              sys.exit("Not a CSV file")

          try:
              with open(sys.argv[1], "r") as file:
                  reader = csv.reader(file)
                  table = []
                  for row in reader:
                      table.append(row)
                  headers = table[0]
                  print(tabulate(table[1:], headers, tablefmt="grid"))
          except FileNotFoundError:
              sys.exit("File does not exist")

      main()
      ```
    - 关键点：由于`tabulate`需要列表，所有读取csv文件时采用了`reader`而不是字典读。
    - 同时刚好可以将读出的第一行即标题作为`header`传给`tabulate`
- `scourgify.py`
    - ```
      import sys
      import csv

      def main():
          if len(sys.argv) < 2:
              sys.exit("Too few command-line arguments")
          elif len(sys.argv) > 3:
              sys.exit("Too many command-line arguments")

          if not sys.argv[1].endswith(".csv"):
              sys.exit("Not a CSV file")

          students = []

          try:
              with open(sys.argv[1], "r") as file:
                  reader = csv.DictReader(file)
                  for row in reader:
                      students.append({"name": row["name"], "house": row["house"]})
          except FileNotFoundError:
              sys.exit(f"Could not read {sys.argv[1]}")

          with open(sys.argv[2], "w") as file:
              writer = csv.DictWriter(file, fieldnames=["first", "last", "house"])
              writer.writeheader()
              for student in students:
      ```

```
            last, first = student["name"].split(", ")
            writer.writerow({"first": first, "last": last, "house":
student["house"]})

main()
```

- 关键点：暂时用一个`list`存储读出的`dict`

- `shirt.py`

```
import sys
import os
from PIL import Image, ImageOps

def main():
    if len(sys.argv) < 2:
        sys.exit("Too few command-line arguments")
    elif len(sys.argv) > 3:
        sys.exit("Too many command-line arguments")

    src = sys.argv[1].lower()
    dst = sys.argv[2].lower()

    if not os.path.splitext(src)[1] in [".jpg", ".jpeg", ".png"]:
        sys.exit("Invalid input")
    if not os.path.splitext(dst)[1] in [".jpg", ".jpeg", ".png"]:
        sys.exit("Invalid output")
    if os.path.splitext(src)[1] != os.path.splitext(dst)[1]:
        sys.exit("Input and output have different extensions")
    try:
        with Image.open(src) as image:
            shirt = Image.open("shirt.png")
            image = ImageOps.fit(image, shirt.size)
            image.paste(shirt, shirt)
            image.save(dst)
            shirt.close()
    except FileNotFoundError:
        sys.exit("Input does not exist")


main()
```

- 关键点：对于文档的阅读，能去找寻对应的方法。

Regular Expressions

正则表达式

- 允许我们检查代码中的模式。例如，验证电子邮件地址的格式是否正确。
- 基础示例
  - 以下是一个简单的Python程序，用于验证电子邮件地址是否包含`@`符号。

```
email = input("What's your email? ").strip()
if "@" in email:
    print("Valid")
else:
    print("Invalid")
```

    - `strip()`方法用于移除输入字符串的首尾空白字符。
- 改进验证
    - 仅检查`@`符号是不够的，我们需要更精确的验证。

```
email = input("What's your email? ").strip()
if "@" in email and "." in email:
    print("Valid")
else:
    print("Invalid")
```

    - 进一步改进，通过分割字符串来验证用户名和域名。

```
email = input("What's your email? ").strip()
username, domain = email.split("@")
if username and "." in domain:
    print("Valid")
else:
    print("Invalid")
```

    - 但显然上述代码仍有漏洞，我们可以使用内置库`re`
- **使用正则表达式库**
    - Python的`re`库提供了内置函数来根据模式验证用户输入。

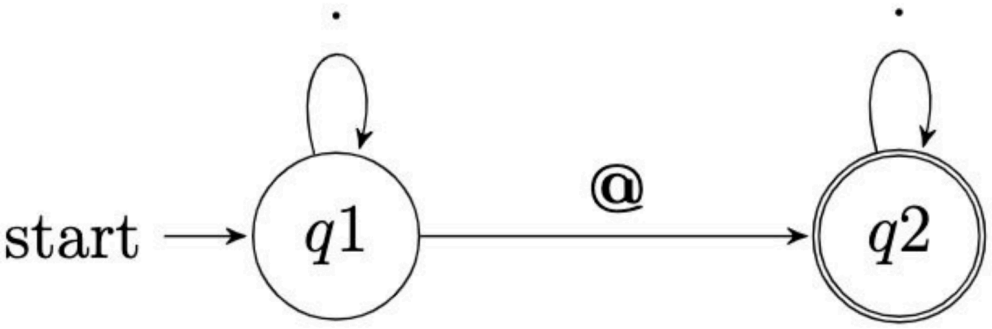
```
import re
email = input("What's your email? ").strip()
if re.search("@", email):
    print("Valid")
else:
    print("Invalid")
```

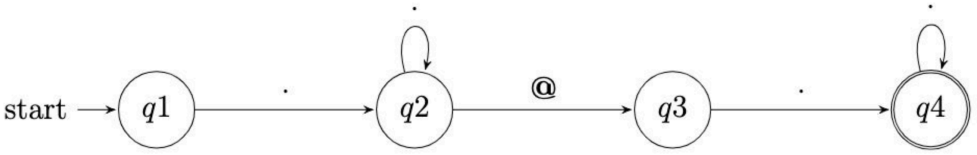
    - 最多用法的方法`search`，使用方法`re.search(pattern, string, flags=0)`
    - 但事实上上述代码并未使得我们的代码功能性更强，我们还需了解特殊符号。
- 特殊符号
    - 在正则表达式中，特殊符号可以用来识别模式。
    - `.`: 除换行符之外的任意字符
    - `*`: 0次或多次重复
    - `+`: 1次或多次重复
    - `?`: 0次或1次重复
    - `{m}`: **m次重复**
    - `{m,n}`: m到n次重复
- 完善电子邮件验证
    -

```
import re

email = input("What's your email? ").strip()

if re.search(".+@.+", email):
    print("Valid")
else:
    print("Invalid")
```

- 

    - `.`用以匹配任何字符，`.*@.*`可以用上述状态转换模型来表示。
    - 

    - `.+@.+`可用上面的状态机转换模型表示，`+`表示至少匹配一个字符。
    - 切记，单独的`.`表示匹配任何字符，`\.`表示匹配`.`这个字符。
- `raw strings`和`regular strings`
    - 在字符串前加上`r`表示将字符串视为`raw`，即不存在特殊字符。
        - ```
          import re

          email = input("What's your email? ").strip()

          if re.search(r"^.+@.+\.edu$", email):
              print("Valid")
          else:
              print("Invalid")
          ```
        - 上述代码会将`\.`视为`\`和`.`两个字符，而这在正则中表示匹配一个`.`。
- 更多特殊符号
    - `^`从字符串开始的位置匹配。
    - `$`匹配字符串结尾或者字符串新行前的末尾位置。
    - `[]`表示字符集合。
    - `[^]`表示补集，即不包含该元素的字符组成的集合。例如`[^@]+`表示除了`@`以外的任何字符。
- 更进一步完善
    - ```
      import re

      email = input("What's your email? ").strip()

      if re.search(r"^[a-zA-Z0-9_]+@[a-zA-Z0-9_]+\.edu$", email):
          print("Valid")
      else:
          print("Invalid")
      ```

        - `[a-zA-Z0-9_]` 表示字符必须满足 `a-z` 或者 `A-Z` 或者 `0-9` 或者 `_`，其余都不行。
    - 以下是一个更完善的电子邮件验证示例。

```
import re
email = input("What's your email? ").strip()
if re.search(r"^\w+@\w.+\.(com|edu|gov|net|org)$", email):
    print("Valid")
else:
    print("Invalid")
```

    - 这里的 `\w` 等同于 `[a-zA-Z0-9_]`。
    - `|` 相当于 `or`

- **其他模式**
    - `\d`：十进制数字
    - `\D`：非十进制数字
    - `\s`：空白字符
    - `\S`：非空白字符
    - `\w`：单词字符，包括数字和下划线
    - `\W`：非单词字符
- 分组，用来配合 `matches` 使用，获取分组。
    - `A|B`：A或B
    - `(...)`：分组
    - `(?:...)`：非捕获分组

大小写敏感性

- 默认情况下，正则表达式是大小写敏感的。
- 要执行大小写不敏感的搜索，请使用 `re.IGNORECASE` 标志。

```
import re

email = input("What's your email? ").strip()

if re.search(r"^\w+@(\w+\.)?\w+\.edu$", email, re.IGNORECASE):
    print("Valid")
else:
    print("Invalid")
```

- `(\w+\.)?` 表示没有或者只出现一次。

清理用户输入

- 用户可能不会总是按照您的期望输入数据。以下是一些清理数据的方法。
- 在终端窗口中输入 `code format.py`。然后在文本编辑器中编写以下代码：

```
name = input("What's your name? ").strip()
print(f"hello, {name}")
```

- 如果用户输入的是"Malan, David"，程序可能不会按预期工作。我们如何修改程序来清理这个输入呢？

```
import re

name = input("What's your name? ").strip()
matches = re.search(r"^(.+), (.+)$", name)
if matches:
    last, first = matches.groups()
    name = first + " " + last
print(f"hello, {name}")
```

- 简化写法`search`和`match`
    - ```
      import re

      name = input("What's your name? ").strip()
      if matches := re.search(r"^(.+), *(.+)$", name):
          name = matches.group(2) + " " + matches.group(1)
      print(f"hello, {name}")
      ```

提取用户输入

- 到目前为止，我们已经验证了用户的输入并清理了用户的输入。现在，让我们从用户输入中提取一些特定信息。
- 使用`replace`方法进行替换。
- 使用`removeprefix`方法去掉字符串开头的内容。
- `re`库中的`sub`方法同样可进行替代。
    - ```
      re.sub(pattern, repl, string, count=0, flags=0)
      ```
    - ```
      import re

      url = input("URL: ").strip()

      username = re.sub(r"https://twitter.com/", "", url)
      print(f"Username: {username}")
      ```
- 在终端窗口中输入`code twitter.py`，然后在文本编辑器窗口中编写以下代码：

```
url = input("URL: ").strip()
if matches := re.search(r"^https?://(?:www\.)?twitter\.com/([a-z0-9_]+)", url,
re.IGNORECASE):
    print(f"用户名: {matches.group(1)}")
```

    - `?:`表示不一定需要捕获。

课外Shorts

- **Patterns**
    - 颜色的匹配，例#0076BA
    - ```
      import re


      def main():
          code = input("Hexadecimal color code: ")
      ```

```
    pattern = r"#"
    match = re.search(pattern, code)
    if match:
        print(f"Valid. Matched with {match.group()}")
    else:
        print("Invalid")


main()
```

- 采用search根据模式进行寻找匹配。
- 锚点：告知匹配应该从开头开始或结尾之内。

```
import re


def main():
    code = input("Hexadecimal color code: ")

    pattern = r"^#[a-fA-F0-9]{6}$"
    match = re.search(pattern, code)
    if match:
        print(f"Valid. Matched with {match.group()}")
    else:
        print("Invalid")


main()
```

- **Capture Groups**
  - 提取，使用正则表达式和捕获组来**动态**捕获正在寻找的内容。使用`()`包裹。
  - 注意捕获`group`中的索引从1开始。

```
import re


locations = {"+1": "United States and Canada", "+62": "Indonesia", "+505":
"Nicaragua"}


def main():
    pattern = r"(?P<country_code>\+\d{1,3}) \d{3}-\d{3}-\d{4}"
    number = input("Number: ")

    match = re.search(pattern, number)
    if match:
        country_code = match.group("country_code")
        print(locations[country_code])
    else:
        print("Unknown")


main()
```

- 更有效的做法，为捕获组起别名并进行访问。

Problem Set 7

- `numb3rs.py`
  - ```
    import re

    def main():
        print(validate(input("IPv4 Address: ")))

    def validate(ip):
        pattern = r"^(\w+)\.(\w+)\.(\w+)\.(\w+)$"
        match = re.search(pattern, ip)
        if match:
            first, second, third, forth = match.groups()
            if judge(first) and judge(second) and judge(third) and judge(forth):
                return True
        return False

    def judge(number):
        if 0 <= int(number) <= 255:
            return True
        return False


    if __name__ == "__main__":
        main()
    ```
  - 注意捕获组每个`()`只会捕获匹配的最后一个。
  - 对应测试`test_num3rs.py`
  - ```
    from numb3rs import validate


    def test_true():
        assert validate("127.0.0.1") == True

    def test_false_01():
        assert validate("256.255.255.255") == False
        assert validate("100.256.256.256") == False

    def test_false_02():
        assert validate("cat") == False
    ```
- `watch.py`
  - ```
    import re

    def main():
        print(parse(input("HTML: ")))


    def parse(s):
        pattern = r"<iframe.+</iframe>"
        match = re.search(pattern, s)
        if match:
            iframe = match.group()
            pattern = r"src=\"https?://(www.)?youtube\.com/embed/([^\"]*)\""
            match = re.search(pattern, iframe)
            if match:
                url = match.group(2)
    ```
    ```

```
            return "https://youtu.be/" + url
    return None


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

- 关键点：正则表达式模式的书写，关键点`.`、`"`等的注意。

- `working.py`

```
import re
import sys


def main():
    print(convert(input("Hours: ")))


def convert(s):
    pattern = r"^(\S+) (AM|PM) to (\S+) (AM|PM)$"
    match = re.search(pattern, s)
    if match:
        first = match.group(1)
        sign_f = match.group(2)
        second = match.group(3)
        sign_s = match.group(4)
        hour_f, min_f = validate(first)
        hour_s, min_s = validate(second)

        first = f"{cal(hour_f, sign_f):02}:{int(min_f):02}"
        second = f"{cal(hour_s, sign_s):02}:{int(min_s):02}"

        return f"{first} to {second}"
    else:
        raise ValueError


def validate(time):
    if ":" in time:
        tmp = time.split(":")
        if not (0 <= int(tmp[0]) <= 12 and 0 <= int(tmp[1]) < 60):
            raise ValueError
        else:
            return (tmp[0], tmp[1])
    else:
        return (time, "00")

def cal(hour, sign):
    if sign == "PM":
        return int(hour) % 12 + 12
    else:
        return int(hour) % 12



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

- 关键点：正则模式，捕获组，条件判断。
- 测试 `test_working.py`
  - ```
    from working import convert
    import pytest

    def test_error():
        with pytest.raises(ValueError):
            convert("9:00 AM 5:00 PM")
        with pytest.raises(ValueError):
            convert("14:00 AM to 5:00 PM")

    def test_success():
        assert convert("9:00 AM to 5:00 PM") == "09:00 to 17:00"
        assert convert("9 AM to 5 PM") == "09:00 to 17:00"
    ```

- `um.py`
  - ```
    import re
    import sys


    def main():
        print(count(input("Text: ")))


    def count(s):
        pattern = r"\bum\b"
        matches = re.findall(pattern, s.lower())
        return len(matches)



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

  - 关键点：`\b` 用以筛选单词本身，`findall` 返回 `list` 故可获取其 `len`
  - 测试 `test_um.py`
    - ```
      from um import count

      def test_0():
          assert count("yummy") == 0

      def test_1():
          assert count("um?") == 1

      def test_2():
          assert count("Um, thanks, um...") == 2
      ```

- `response.py`
      ```

    - ```
      from validator_collection import validators, errors

      address = input("What's your email address? ")

      try:
          email_address = validators.email(address)
          # Will raise an EmptyValueError
      except errors.EmptyValueError:
          print("Invalid")
      except errors.InvalidEmailError:
          print("Invalid")
      else:
          print("Valid")
      ```
    - 关键点：调包进行邮件验证即可。

Object-Oriented Programming

简介

- 编程中有不同的范式，面向对象编程（OOP）是解决编程问题的强大方案。
- OOP与之前的过程化、逐步编程范式不同。

过程化编程示例step-by-step

- 输入姓名和学院，然后打印出来。
- 示例代码：

```
name = input("Name: ")
house = input("House: ")
print(f"{name} from {house}")
```

抽象与函数

- 使用函数抽象程序的某些部分，例如：

```
def main():
    name = get_name()
    house = get_house()
    print(f"{name} from {house}")

def get_name():
    return input("Name: ")

def get_house():
    return input("House: ")

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

    - 方法从`main`中抽象出来，同时底部告诉解释器去运行`main`方法。
- Python可以返回多个返回值
```

    - ```
      def main():
          name, house = get_student()
          print(f"{name} from {house}")


      def get_student():
          name = input("Name: ")
          house = input("House: ")
          return name, house


      if __name__ == "__main__":
          main()
      ```
- 也可以将值打包为元组形式
    - ```
      def main():
          student = get_student()
          print(f"{student[0]} from {student[1]}")


      def get_student():
          name = input("Name: ")
          house = input("House: ")
          return (name, house)


      if __name__ == "__main__":
          main()
      ```
    - 访问元组元素即可，注意元组内元素不可改变。
- 若要使得元素可改变，可打包为数组形式。
    - ```
      def main():
          student = get_student()
          if student[0] == "Padma":
              student[1] = "Ravenclaw"
          print(f"{student[0]} from {student[1]}")


      def get_student():
          name = input("Name: ")
          house = input("House: ")
          return [name, house]


      if __name__ == "__main__":
          main()
      ```
- 甚至返回值可以是一个字典。字典也是可变的。
    - ```
      def main():
          student = get_student()
          print(f"{student['name']} from {student['house']}")


      def get_student():
          student = {}
      ```

```
    student["name"] = input("Name: ")
    student["house"] = input("House: ")
    return student


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

数据结构

- 使用元组（tuple）返回多个值，但元组是不可变的。
- 使用列表（list）提供灵活性，但列表是可变的，可能会导致错误。
- 使用字典（dictionary）通过键值对返回数据，提供了一种更清晰的数据访问方式。

类（Class）

- 类是OOP中创建自定义数据类型的方式。
- 示例代码：

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

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)
```

方法（Method）

- 在类内部定义的函数称为方法。
- `__init__` 方法是构造函数，用于初始化对象。`self`表示的是刚刚创建的对象。
- `__str__` 方法用于定义对象的字符串表示。
    - 也是伴随类的内置方法。可以打印对象、属性以及与对象有关的几乎所有内容。

异常处理

- 使用 `raise` 关键字抛出自定义异常。
- 示例代码：

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Missing name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house
```

自定义方法

- 在**类中**定义自定义方法，例如 `charm` 方法。
- 示例代码：

```
def charm(self):
    match self.patronus:
        case "Stag":
            return "🦌"
        case "Otter":
            return "🦦"
        case "Jack Russell terrier":
            return "🐶"
        case _:
            return "🦊•🪄"
```

装饰器（Decorators）

- 使用属性（Properties）可以加强代码。
- 在Python中，通过函数装饰器定义属性，以`@`开头。
- 示例代码：

```
@property
def house(self):
    return self._house
```

- `@property` 定义了一个属性的getter方法。
- `@house.setter` 定义了一个属性的setter方法，用于在设置属性值时进行验证。
- 示例代码：
  - ```
    # Setter for house
    @house.setter
    def house(self, house):
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self._house = house
    ```
  - 例如在 `student.house = "Gryffindor"` 时就会调用。
  - 为什么使用 `_house` 而不是 `house`？
    - `house` 是我们类的属性，用来改变类的属性。
    - `_house` 是属性本身，`_`表明用户不需要也不应该直接修改该值。`_house` 应该只能由 `house` 这个setter来设置。

联系之前

- `int`、`str`、`list`、`dict` 都是类。

类方法（Class Methods）

- 有时，我们希望向类本身添加功能，而不是类的实例。
- `@classmethod` 是一个装饰器，用于向整个类添加功能。
- 示例代码：
    ```

```
@classmethod
def sort(cls, name):
    print(name, "is in", random.choice(cls.houses))
```

- 类方法中用`cls`代替`self`。
- 示例代码：
  - ```
    import random


    class Hat:

        houses = ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]

        @classmethod
        def sort(cls, name):
            print(name, "is in", random.choice(cls.houses))


    Hat.sort("Harry")
    ```
  - 由于不需要实例化对象，我们可以移除`__init__`方法。
  - 类方法调用：`类名.方法名(pa...)`

静态方法（Static Methods）

- `@staticmethod`是另一种方法，与类方法和实例方法不同。
- 可以自行探索静态方法及其与类方法的区别。
- 需要访问或修改类变量，应该使用类方法。
- 不需要访问类或实例的状态，或者你的方法只是逻辑上与类相关，但不依赖于类或实例的状态，那么应该使用静态方法。

继承（Inheritance）

- 继承是面向对象编程中最强大的特性之一。
- 可以创建一个类，它"继承"另一个类的方法、变量和属性。
- 示例代码：

```
class Student(Wizard):
    def __init__(self, name, house):
        super().__init__(name)
        self.house = house
```

- `Student`类继承了`Wizard`类的特性。

异常处理与继承

- 异常也构成了一个层次结构，其中包含子类、父类和祖类。
- 示例：

```
BaseException
 +-- KeyboardInterrupt
 +-- Exception
      +-- ArithmeticError
      |    +-- ZeroDivisionError
```
```

```
      +-- AssertionError
      +-- AttributeError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- KeyError
      +-- NameError
      +-- SyntaxError
      |    +-- IndentationError
      +-- ValueError
```

运算符重载（Operator Overloading）

- 一些运算符如`+`和`-`可以被"重载"，从而具有超出简单算术运算的能力。
- 示例代码：

```
def __add__(self, other):
    galleons = self.galleons + other.galleons
    sickles = self.sickles + other.sickles
    knuts = self.knuts + other.knuts
    return Vault(galleons, sickles, knuts)
```

- `__add__`方法允许两个Vault实例相加。

总结

- 面向对象编程提供了新的能力层次。
- 关键概念包括：
  - 类（Classes）
  - 抛出异常（raise）
  - 类方法（Class Methods）
  - 静态方法（Static Methods）
  - 继承（Inheritance）
  - 运算符重载（Operator Overloading）

课外Shorts

- **Classes**
  - 为什么使用类？需要更为稳固的东西来存储。
  - 将**模板**一样的东西封装存放在一起。
  - 类名往往首字母大写。
  - `__init__`方法至少有一个参数`self`表示创建的新对象。
    - 从而为对象的新实例分配传递的内容。
  - ```
    class Package:
        def __init__(self, number, sender, recipient, weight):
            self.number = number
            self.sender = sender
            self.recipient = recipient
            self.weight = weight


    def main():
    ```
    ```

```
    packages = [
        Package(number=1, sender="Alice", recipient="Bob", weight=10),
        Package(number=2, sender="Bob", recipient="Charlie", weight=5),
    ]


main()
```

- **Class Method and Class Variables**
  - 属于类的方法?
    - `@classmethod`装饰器进行修饰。
    - `cls`作为第一个参数表示类本身。
    - 类方法甚至可用来创建实例。
      - ```
        class Food:
            base_hearts = 1

            def __init__(self, ingredients):
                self.ingredients = ingredients
                self.hearts = Food.calculate_hearts(ingredients)

            @classmethod
            def calculate_hearts(cls, ingredients):
                hearts = cls.base_hearts
                for ingredient in ingredients:
                    if "hearty" in ingredient.lower():
                        hearts += 2
                    else:
                        hearts += 1
                return hearts

            @classmethod
            def from_nothing(cls, hearts):
                food = cls(ingredients=[])
                food.hearts = hearts
                return food


        def main():
            mushroom_skewer = Food(ingredients=["Mushroom", "Hearty
        Mushroom"])
            print(f"This Mushroom Skewer heals {mushroom_skewer.hearts}
        hearts!")

            Food.base_hearts = 2
            mushroom_skewer = Food(ingredients=["Mushroom", "Hearty
        Mushroom"])
            print(f"This Mushroom Skewer heals {mushroom_skewer.hearts}
        hearts!")

            mushroom_skewer = Food.from_nothing(hearts=2)
            print(f"This Mushroom Skewer heals {mushroom_skewer.hearts}
        hearts!")


        main()
        ```
        ```

        - `from_nothing`类方法返回了一个`food`对象，所以我们可以使用类方法创建实例。
    - 类变量
        - 要在所有实例上作用，所有实例之间共享的变量。
        - 一般定义在`init`上方。
        - 如何访问？`cls.var`，`cls`表示类本身。
- **Instance Variables**
    - `self.xxx`属于实例本身，称为实例变量。
    - 访问实例变量使用`.`
    - ```
      for package in packages:
          print(f"{package.number}: {package.sender} to {package.recipient},
      {package.weight}kg")
      ```
- **Instance Methods**
    - 可以在任意特定实例上运行的方法。
    - `self`是每个实例方法隐含的第一个变量。
    - `__str__`方法，`__`常常表明特殊方法，`str`会在`print`方法被调用。
    - 访问实例方法使用`.`

Problem Set 8

- `seasons.py`
    - ```
      from datetime import date
      import inflect
      import re
      import sys


      def main():
          pattern = r"^([0-9]{4})-([0-9]{2})-([0-9]{2})$"
          birthday = input("Date of Birth: ")
          match = re.search(pattern, birthday)
          if match:
              time = date(int(match.group(1)), int(match.group(2)),
      int(match.group(3)))
              print(cal(time))
          else:
              sys.exit("Invalid date")

      def cal(time):
          minutes = date.today() - time
          p = inflect.engine()
          result = f"{p.number_to_words(minutes.days * 24 * 60)} minutes"
          result = result[0].upper() + result[1:]
          return result.replace(" and", "")

      if __name__ == "__main__":
          main()
      ```
    - 关键点：正则提取出需要的元素，好像跟OOP也没啥关系吧。
- `jar.py`
    - ```
      class Jar:
      ```
      ```

```
    def __init__(self, capacity=12):
        if capacity < 0:
            raise ValueError
        else:
            self.capacity = capacity
            self.size = 0

    def __str__(self):
        return self.size * "🍪"

    def deposit(self, n):
        if (n + self.size) > self.capacity:
            raise ValueError
        else:
            self.size += n

    def withdraw(self, n):
        self.size -= n
        if self.size < 0:
            raise ValueError

    # @property
    def capacity(self):
        return self.capacity

    # @property
    def size(self):
        return self.size
```

- 由于加上`property`等注解还需要`setter`，故我注释掉了。
- 测试`test_jar.py`
- ```
  from jar import Jar
  import pytest


  def test_init():
      jar = Jar(10)
      assert str(jar) == ""
      assert jar.capacity == 10
      assert jar.size == 0


  def test_str():
      jar = Jar()
      assert str(jar) == ""
      jar.deposit(1)
      assert str(jar) == "🍪"
      jar.deposit(11)
      assert str(jar) == "🍪🍪🍪🍪🍪🍪🍪🍪🍪🍪🍪🍪"


  def test_deposit():
      jar = Jar(2)
      jar.deposit(1)
      assert jar.size == 1
      with pytest.raises(ValueError):
          jar.deposit(10)
  ```
  ```

```
def test_withdraw():
    jar = Jar(10)
    jar.deposit(10)
    jar.withdraw(1)
    assert jar.size == 9

    with pytest.raises(ValueError):
        jar.withdraw(10)
```

- `shirtificate.py`
  - ```
    from fpdf import FPDF

    name = input("Name: ")

    pdf = FPDF(orientation="P", unit="mm", format="A4")
    pdf.add_page()
    pdf.set_auto_page_break(auto=False)

    pdf.set_font('Helvetica', style='B', size=26)
    pdf.cell(200, 10, 'CS50 Shirtificate', ln=True, align='C')

    pdf.image('shirtificate.png', x=(210 - 150) / 2, w=150)

    pdf.set_font('Arial', 'B', 24)
    pdf.set_text_color(255, 255, 255)
    pdf.cell(200, -80, f"{name} took CS50", ln=True, align='C')

    # 保存 PDF
    pdf.output('shirtificate.pdf')
    ```
  - 答案来自于智谱。

Et Cetera

集合（set）

- 集合用于存储不重复的元素。
- **自动去重**，无需编写额外代码。
- 示例代码：

```
students = [...]
houses = set()
for student in students:
    houses.add(student["house"])
for house in sorted(houses):
    print(house)
```

全局变量

- 在函数外部定义的变量，可以在函数内部使用。
- 使用 `global` 关键字在函数内部声明全局变量。
- 示例代码：
```

```
balance = 0
def main():
    global balance
    # 使用全局变量
```

- 可以借助于面向对象而不是全局变量的方式解决问题。
    - ```
      class Account:
          def __init__(self):
              self._balance = 0

          @property
          def balance(self):
              return self._balance

          def deposit(self, n):
              self._balance += n

          def withdraw(self, n):
              self._balance -= n


      def main():
          account = Account()
          print("Balance:", account.balance)
          account.deposit(100)
          account.withdraw(50)
          print("Balance:", account.balance)


      if __name__ == "__main__":
          main()
      ```
    - 如此创建一个实例即可在各个位置可以修改获得。
    - 谨慎使用全局变量！

常量

- 用大写字母命名的变量，约定为常量，不应被修改。
- 示例代码：

```
MEOWS = 3
for _ in range(MEOWS):
    print("meow")
```

- 实际不能保证无法被修改。

类型提示（Type Hints）

- 提供了函数参数和返回值的类型信息。
- 使用 `mypy` 工具检查类型错误。
- 示例代码：
```

```python
def meow(n: int) -> None:
    for _ in range(n):
        print("meow")

number: int = int(input("Number: "))
meow(number)
```

文档字符串（Docstrings）

- 用于描述函数的目的和用法，是一种标准的注释写法。
- 可使用例如`Sphinx`等工具解析docstrings并自动创建文档。
- 示例代码：

```python
def meow(n):
    """
    Meow n times.
    :param n: Number of times to meow
    :type n: int
    :return: A string of n meows, one per line
    :rtype: str
    """
    return "meow\n" * n
```

argparse

- `argparse`为库，用于解析命令行参数。
- 提供帮助信息和默认值。
- 示例代码：

```python
import argparse
parser = argparse.ArgumentParser(description="Meow like a cat")
parser.add_argument("-n", default=1, help="number of times to meow", type=int)
args = parser.parse_args()
for _ in range(args.n):
    print("meow")
```

  - `help`可以帮助用户了解如何使用这个程序。
  - `default`可以在用户未提供参数时提供默认值。

打包（Unpacking）

- 从单个变量中分割出两个变量：

```python
first, _ = input("What's your name? ").split(" ")
print(f"hello, {first}")
```

- 使用`list`及下标。
- 使用`*`操作符解包列表，将**列表元素**作为函数参数：

```python
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

coins = [100, 50, 25]
print(total(*coins), "Knuts")
```

- 当要用到名称和值时，字典应该被想到。

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts


coins = {"galleons": 100, "sickles": 50, "knuts": 25}

print(total(coins["galleons"], coins["sickles"], coins["knuts"]), "Knuts")
```

- 使用 `**` 操作符解包字典，将字典的**键值**对作为函数参数：

```
coins = {"galleons": 100, "sickles": 50, "knuts": 25}
print(total(**coins), "Knuts")
```

args 和 kwargs

- `*args` 用于收集位置参数，`**kwargs` 用于收集关键字参数：

```
def f(*args, **kwargs):
    print("Positional:", args)
    print("Named:", kwargs)
f(100, 50, 25)
# Positional: (100, 50, 25)
# Named: {}
f(galleons=100, sickles=50, knuts=25)
# Positional: ()
# Named: {'galleons': 100, 'sickles': 50, 'knuts': 25}
```

map

- `map` 函数可以**将一个函数**应用于一个序列的**每个元素**：

```
def yell(*words):
    uppercased = map(str.upper, words)
    print(*uppercased)

yell("This", "is", "CS50")
```

  - 结合前文，使用 `*` 解包 `list`
  - `*words` 允许函数接受很多参数。
  - `map` 接收两个参数，要使用的函数以及将要作用的 `list`。

列表推导式（List Comprehensions）

- 使用列表推导式创建列表，提高代码的可读性和效率：

```
def yell(*words):
    uppercased = [arg.upper() for arg in words]
    print(*uppercased)

yell("This", "is", "CS50")
```

  - 每一个量都使用 `upper` 作用在上面。

- 同样可作用于存储字典的列表上

```
students = [
    {"name": "Hermione", "house": "Gryffindor"},
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Ron", "house": "Gryffindor"},
    {"name": "Draco", "house": "Slytherin"},
]

gryffindors = [
    student["name"] for student in students if student["house"] == "Gryffindor"
]

for gryffindor in sorted(gryffindors):
    print(gryffindor)
```

  - 极大简化代码。

filter

- `filter` 函数可以基于条件返回一个序列的**子集**：

```
students = [{"name": "Hermione", "house": "Gryffindor"}, ...]
gryffindors = filter(lambda s: s["house"] == "Gryffindor", students)
for gryffindor in sorted(gryffindors, key=lambda s: s["name"]):
    print(gryffindor["name"])
```

  - 需要一个 `filtering` 函数进行过滤，返回布尔值。
  - `filter` 函数接收两个参数，第一个是要作用在每个元素的函数，第二个是将作用的序列。

字典推导式（Dictionary Comprehensions）

- 使用字典推导式创建字典，提高代码的简洁性：

```
students = ["Hermione", "Harry", "Ron"]
gryffindors = {student: "Gryffindor" for student in students}
print(gryffindors)
```

enumerate

- 原先实现：
  - ```
    students = ["Hermione", "Harry", "Ron"]

    for i in range(len(students)):
        print(i + 1, students[i])
    ```

- `enumerate` 函数可以用来获取序列的索引和值：

```
students = ["Hermione", "Harry", "Ron"]
for i, student in enumerate(students):
    print(i + 1, student)
```

生成器和迭代器（Generators and Iterators）

- 使用 `yield` 关键字创建生成器，可以逐个产生值而不是一次性产生整个列表：
```

```
def sheep(n):
    for i in range(n):
        yield "🐑" * i
for s in sheep(1000000):
    print(s)
```

- `yield`每次只提供一个值，减小内存负担。

恭喜!

- 你已经完成了 CS50 的学习，掌握了许多编程工具和技巧。
- 最后一个程序示例：

```
import cowsay
import pyttsx3
engine = pyttsx3.init()
this = input("What's this? ")
cowsay.cow(this)
engine.say(this)
engine.runAndWait()
```

- 希望你能利用所学知识解决现实世界的问题，让世界变得更美好。

课外Shorts

- **Recursion**
  - 纸牌的排列组合——阶乘。
  - 更简单的方法定义阶乘？`3! = 3 * 2!`，`n! = n * (n - 1)`
  - 递归应该包括递归式和基本情况及确切知道答案的情况。

Final Project：

- maybe one day in the future🤣