Lecture 9 第 9 讲

Et Cetera 等等

- Over the many past lessons, we have covered so much related to Python!

  在过去的许多课程中，我们已经介绍了许多与 Python 相关的内容！
- In this lesson, we will be focusing upon many of the "et cetera" items not previously discussed. "Et cetera" literally means "and the rest"!

  在本课中，我们将重点介绍许多之前未讨论的 "et cetera" 项目。"Et cetera" 的字面意思是 "和其余的"！
- Indeed, if you look at the Python documentation, you will find quite "the rest" of other features.

  事实上，如果你看一下 Python 文档，你会发现其他功能的 "其余" 功能相当 "其余"。

`set` 设置

- In math, a set would be considered a set of numbers without any duplicates.

  在数学中，一组将被视为没有任何重复项的一组数字。
- In the text editor window, code as follows:

  在文本编辑器窗口中，编写如下代码：

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

houses = []
for student in students:
    if student["house"] not in houses:
        houses.append(student["house"])

for house in sorted(houses):
    print(house)
```

  Notice how we have a list of dictionaries, each being a student. An empty list called `houses` is created. We iterate through each `student` in `students`. If a student's `house` is not in `houses`, we append to our list of `houses`.

  请注意我们是如何有一个词典列表的，每个词典都是一个学生。将创建一个名为 `houses` 的空列表。我们在 `students` 中遍历每个 `student`。如果学生的 `房子` 不在 `房子` 里，我们会在我们的 `房子` 列表中附加。
- It turns out we can use the built-in `set` features to eliminate duplicates.

  事实证明，我们可以使用内置的 `set` 功能来消除重复项。
- In the text editor window, code as follows:

  在文本编辑器窗口中，编写如下代码：

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

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

for house in sorted(houses):
    print(house)
```

  Notice how no checking needs to be included to ensure there are no duplicates. The `set` object takes care of this for us automatically.

  请注意，不需要包含任何检查以确保没有重复项。`set` 对象会自动为我们处理这个问题。
- You can learn more in Python's documentation of `set`.

  您可以在 Python 的 `set` 文档中了解更多信息。

Global Variables 全局变量

- In other programming languages, there is the notion of global variables that are accessible to any function.

在其他编程语言中，存在任何函数都可以访问的全局变量的概念。

- We can leverage this ability within Python. In the text editor window, code as follows:

  我们可以在 Python 中利用此功能。在文本编辑器窗口中，编写如下代码：

```
balance = 0


def main():
    print("Balance:", balance)


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

Notice how we create a global variable called `balance`, outside of any function.

请注意我们如何在任何函数之外创建一个名为 `balance` 的全局变量。

- Since no errors are presented by executing the code above, you'd think all is well. However, it is not! In the text editor window, code as follows:

  由于执行上面的代码没有出现任何错误，因此您会认为一切都很好。然而，事实并非如此！在文本编辑器窗口中，编写如下代码：

```
balance = 0


def main():
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)


def deposit(n):
    balance += n


def withdraw(n):
    balance -= n


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

Notice how we now add the functionality to add and withdraw funds to and from `balance`. However, executing this code, we are presented with an error! We see an error called `UnboundLocalError`. You might be able to guess that, at least in the way we've currently coded `balance` and our `deposit` and `withdraw` functions, we can't reassign it a value value inside a function.

请注意我们现在如何添加向 余额 添加和提取资金的功能。但是，执行此代码时，我们会看到一个错误！我们看到一个名为 `UnboundLocalError` 的错误。你可能能够猜到，至少按照我们目前对 余额 以及 `deposit` 和 `withdraw` 函数的编码方式，我们不能在函数中重新为其分配一个 value 值。

- To interact with a global variable inside a function, the solution is to use the `global` keyword. In the text editor window, code as follows:

  要与函数内的全局变量交互，解决方案是使用 `global` 关键字。在文本编辑器窗口中，编写如下代码：

```
balance = 0


def main():
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)


def deposit(n):
    global balance
    balance += n


def withdraw(n):
    global balance
    balance -= n


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

Notice how the `global` keyword tells each function that `balance` does not refer to a local variable: instead, it refers to the global variable we originally placed at the top of our code. Now, our code functions!

请注意 `global` 关键字如何告诉每个函数 `balance` 不引用局部变量：相反，它引用我们最初放置在代码顶部的全局变量。现在，我们的代码开始运行！

- Utilizing our powers from our experience with object-oriented programming, we can modify our code to use a class instead of a global variable. In the text editor window, code as follows:

  利用我们在面向对象编程方面的经验，我们可以修改我们的代码以使用类而不是全局变量。在文本编辑器窗口中，编写如下代码：

```
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()
```

Notice how we use `account = Account()` to create an account. Classes allow us to solve this issue of needing a global variable more cleanly because these instance variables are accessible to all the methods of this class utilizing `self`.

注意我们如何使用 `account = Account ()` 来创建账户。类允许我们更清晰地解决需要全局变量的问题，因为这些实例变量可以被这个类的所有方法使用 `self` 来访问。

- Generally speaking, global variables should be used quite sparingly, if at all!

  一般来说，全局变量应该非常谨慎地使用，如果有的话！

Constants 常数

- Some languages allow you to create variables that are unchangeable, called "constants". Constants allow one to program defensively and reduce the opportunities for important values to be altered.

  某些语言允许您创建不可更改的变量，称为“常量”。常量允许人们进行防御性编程并减少更改重要值的机会。

- In the text editor window, code as follows:

  在文本编辑器窗口中，编写如下代码：

```
MEOWS = 3

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

  Notice `MEOWS` is our constant in this case. Constants are typically denoted by capital variable names and are placed at the top of our code. Though this *looks* like a constant, in reality, Python actually has no mechanism to prevent us from changing that value within our code! Instead, you're on the honor system: if a variable name is written in all caps, just don't change it!

  请注意，在这种情况下，`MEOWS` 是我们的常数。常量通常由大写变量名称表示，并放置在代码的顶部。虽然这*看起来像*一个常量，但实际上，Python 实际上没有机制来阻止我们在代码中更改该值！相反，你是在 honor 系统上：如果变量名称全部大写，就不要改变它！

- One can create a class "constant", now in quotes because we know Python doesn't quite support "constants". In the text editor window, code as follows:

  可以创建一个类“constant”，现在用引号括起来，因为我们知道 Python 不太支持“constants”。在文本编辑器窗口中，编写如下代码：

```
class Cat:
    MEOWS = 3

    def meow(self):
        for _ in range(Cat.MEOWS):
            print("meow")


cat = Cat()
cat.meow()
```

  Because `MEOWS` is defined outside of any particular class method, all of them have access to that value via `Cat.MEOWS`.

  由于 `MEOWS` 是在任何特定类方法之外定义的，因此它们都可以通过 `Cat.MEOWS` 访问该值。

Type Hints 类型提示

- In other programming languages, one expresses explicitly what variable type you want to use.

  在其他编程语言中，可以显式表示要使用的变量类型。

- As we saw earlier in the course, Python does not require the explicit declaration of types.

  正如我们在课程前面看到的，Python 不需要显式声明类型。

- Nevertheless, it's good practice need to ensure all of your variables are of the right type.

  尽管如此，最好还是需要确保所有变量都是正确的类型。

- `mypy` is a program that can help you test to make sure all your variables are of the right type.

  `MyPy` 是一个程序，可以帮助您进行测试以确保所有变量都是正确的类型。

- You can install `mypy` by executing in your terminal window: `pip install mypy`.

  您可以通过在终端窗口中执行来安装 `mypy`：`pip install mypy`。

In the text editor window, code as follows:

在文本编辑器窗口中，编写如下代码：

```
def meow(n):
    for _ in range(n):
        print("meow")


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

You may already see that `number = input("Number: ")` returns a `string`, not an `int`. But `meow` will likely want an `int`!

你可能已经看到 `number = input（“Number： ）”` 返回一个 字符串，而不是 一个 `int`。但是 `meow` 可能需要一个 `int`！

- A type hint can be added to give Python a hint of what type of variable `meow` should expect. In the text editor window, code as follows:

  可以添加类型提示，为 Python 提供 `meow` 应该期望什么类型的变量的提示。在文本编辑器窗口中，编写如下代码：

```
def meow(n: int):
    for _ in range(n):
        print("meow")


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

  Notice, though, that our program still throws an error.

  但请注意，我们的程序仍然会引发错误。

- After installing `mypy`, execute `mypy meows.py` in the terminal window. `mypy` will provide some guidance about how to fix this error.

  安装 `mypy` 后，在终端窗口中执行 `mypy meows.py`。`MyPy` 将提供一些有关如何修复此错误的指导。

- You can annotate all your variables. In the text editor window, code as follows:

  您可以注释所有变量。在文本编辑器窗口中，编写如下代码：

```
def meow(n: int):
    for _ in range(n):
        print("meow")
```

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

Notice how `number` is now provided a type hint.

请注意 `number` 现在是如何提供类型提示的。

- Again, executing `mypy meows.py` in the terminal window provides much more specific feedback to you, the programmer.

  同样，在终端窗口中执行 `mypy meows.py` 可以为您（程序员）提供更具体的反馈。

- We can fix our final error by coding as follows:

  我们可以通过如下编码来修复最终错误：

```
def meow(n: int):
    for _ in range(n):
        print("meow")


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

Notice how running `mypy` how produces no errors because cast our input as an integer.

请注意运行 `mypy` how 不会产生任何错误，因为将我们的输入转换为整数。

- Let's introduce a new error by assuming that `meow` will return to us a string, or `str`. In the text editor window, code as follows:

  让我们通过假设 `meow` 将返回一个字符串或 `str` 来引入一个新的错误。在文本编辑器窗口中，编写如下代码：

```
def meow(n: int):
    for _ in range(n):
        print("meow")


number: int = int(input("Number: "))
meows: str = meow(number)
print(meows)
```

Notice how the `meow` function has only a side effect. Because we only attempt to print "meow", not return a value, an error is thrown when we try to store the return value of `meow` in `meows`.

请注意 `meow` 函数只有一个副作用。因为我们只尝试打印 "meow"，而不返回值，所以当我们尝试将 `meow` 的返回值存储在 `meow` 中时，会引发错误。

- We can further use type hints to check for errors, this time annotating the return values of functions. In the text editor window, code as follows:

  我们可以进一步使用类型提示来检查错误，这次是注释函数的返回值。在文本编辑器窗口中，编写如下代码：

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


number: int = int(input("Number: "))
meows: str = meow(number)
print(meows)
```

Notice how the notation `-> None` tells `mypy` that there is no return value.

请注意表示法 `-> None` 如何告诉 `mypy` 没有返回值。

- We can modify our code to return a string if we wish:

  如果我们愿意，我们可以修改我们的代码以返回一个字符串：

```
def meow(n: int) -> str:
    return "meow\n" * n


number: int = int(input("Number: "))
meows: str = meow(number)
print(meows, end="")
```

Notice how we store in `meows` multiple `str`s. Running `mypy` produces no errors.

注意我们如何在 `meows` 中存储多个 `str`s。运行 `mypy` 不会产生任何错误。

- You can learn more in Python's documentation of Type Hints.

  您可以在 Python 的类型提示文档中了解更多信息。

- You can learn more about `mypy` through the program's own documentation.

  您可以通过该程序自己的文档了解有关 `mypy` 的更多信息。

Docstrings 文档字符串

- A standard way of commenting your function's purpose is to use a docstring. In the text editor window, code as follows:

  注释函数目的的标准方法是使用 docstring。在文本编辑器窗口中，编写如下代码：

```
def meow(n):
    """Meow n times."""
    return "meow\n" * n


number = int(input("Number: "))
meows = meow(number)
print(meows, end="")
```

Notice how the three double quotes designate what the function does.

请注意三个双引号如何指定函数的作用。

- You can use docstrings to standardize how you document the features of a function. In the text editor window, code as follows: s

  您可以使用文档字符串来标准化记录函数功能的方式。在文本编辑器窗口中，编写如下代码： s

```
def meow(n):
    """
    Meow n times.

    :param n: Number of times to meow
    :type n: int
    :raise TypeError: If n is not an int
    :return: A string of n meows, one per line
    :rtype: str
    """
    return "meow\n" * n
```

```
number = int(input("Number: "))
meows = meow(number)
print(meows, end="")
```

Notice how multiple docstring arguments are included. For example, it describes the parameters taken by the function and what is returned by the function.

请注意如何包含多个文档字符串参数。例如，它描述函数采用的参数以及函数返回的内容。

- Established tools, such as Sphinx, can be used to parse docstrings and automatically create documentation for us in the form of web pages and PDF files such that you can publish and share with others.

  成熟的工具，例如 Sphinx，可用于解析文档字符串并以网页和 PDF 文件的形式自动为我们创建文档，以便您可以发布并与他人共享。

- You can learn more in Python's documentation of docstrings.

  您可以在 Python 的文档字符串文档中了解更多信息。

`argparse`

- Suppose we want to use command-line arguments in our program. In the text editor window, code as follows:

  假设我们想在程序中使用命令行参数。在文本编辑器窗口中，编写如下代码：

```
import sys

if len(sys.argv) == 1:
    print("meow")
elif len(sys.argv) == 3 and sys.argv[1] == "-n":
    n = int(sys.argv[2])
    for _ in range(n):
        print("meow")
else:
    print("usage: meows.py [-n NUMBER]")
```

Notice how `sys` is imported, from which we get access to `sys.argv`—an array of command-line arguments given to our program when run. We can use several `if` statements to check whether the use has run our program properly.

请注意 `sys` 是如何导入的，我们从中可以访问 `sys.argv`，`sys.argv` 是运行时提供给程序的命令行参数数组。我们可以使用多个 `if` 语句来检查 use 是否正确运行了我们的程序。

- Let's assume that this program will be getting much more complicated. How could we check all the arguments that could be inserted by the user? We might give up if we have more than a few command-line arguments!

  让我们假设这个程序会变得更加复杂。我们如何检查用户可以插入的所有参数？如果我们有不止几个命令行参数，我们可能会放弃！

- Luckily, `argparse` is a library that handles all the parsing of complicated strings of command-line arguments. In the text editor window, code as follows:

  幸运的是，`argparse` 是一个处理命令行参数复杂字符串解析的库。在文本编辑器窗口中，编写如下代码：

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-n")
args = parser.parse_args()

for _ in range(int(args.n)):
    print("meow")
```

Notice how `argparse` is imported instead of `sys`. An object called `parser` is created from an `ArgumentParser` class. That class's `add_argument` method is used to tell `argparse` what arguments we should expect from the user when they run our program. Finally, running the parser's `parse_args` method ensures that all of the arguments have been included properly by the user.

请注意 `argparse` 是如何导入的，而不是 `sys` 的。名为 `parser` 的对象是从 `ArgumentParser` 类创建的。该类的 `add_argument` 方法用于告诉 `argparse` 当用户运行我们的程序时，我们应该期望从用户那里得到哪些参数。最后，运行解析器的 `parse_args` 方法可确保用户已正确包含所有参数。

- We can also program more cleanly, such that our user can get some information about the proper usage of our code when they fail to use the program correctly. In the text editor window, code as follows:

  我们还可以更干净地编程，这样我们的用户在未能正确使用程序时可以获得有关正确使用代码的一些信息。在文本编辑器窗口中，编写如下代码：

```
import argparse

parser = argparse.ArgumentParser(description="Meow like a cat")
parser.add_argument("-n", help="number of times to meow")
args = parser.parse_args()

for _ in range(int(args.n)):
    print("meow")
```

Notice how the user is provided some documentation. Specifically, a `help` argument is provided. Now, if the user executes `python meows.py --help` or `-h`, the user will be presented with some clues about how to use this program.

请注意如何向用户提供一些文档。具体来说，提供了一个 `help` 参数。现在，如果用户执行 `python meows.py --help` 或 `-h`，用户将看到一些有关如何使用此程序的线索。

- We can further improve this program. In the text editor window, code as follows:

  我们可以进一步改进这个程序。在文本编辑器窗口中，编写如下代码：

```
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")
```

Notice how not only is help documentation included, but you can provide a `default` value when no arguments are provided by the user.

请注意，它不仅包括帮助文档，而且在用户未提供任何参数时，您还可以提供 默认值 。

- You can learn more in Python's documentation of `argparse`.

  您可以在 Python 的 `argparse` 文档中了解更多信息。

Unpacking 打开

- Would it not be nice to be able to split a single variable into two variables? In the text editor window, code as follows:

  能够将单个变量拆分为两个变量不是很好吗？在文本编辑器窗口中，编写如下代码：

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

Notice how this program tries to get a user's first name by naively splitting on a single space.

请注意此程序如何尝试通过天真地拆分单个空格来获取用户的名字。

- It turns out there are other ways to unpack variables. You can write more powerful and elegant code by understanding how to unpack variables in seemingly more advanced ways. In the text editor window, code as follows:

事实证明，还有其他方法可以解压缩变量。通过了解如何以看似更高级的方式解压缩变量，您可以编写更强大、更优雅的代码。在文本编辑器窗口中，编写如下代码：

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


print(total(100, 50, 25), "Knuts")
```

Notice how this returns the total value of Knuts.

请注意这如何返回 Knuts 的总值。

- What if we wanted to store our coins in a list? In the text editor window, code as follows:

如果我们想将硬币存储在列表中怎么办？在文本编辑器窗口中，编写如下代码：

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


coins = [100, 50, 25]

print(total(coins[0], coins[1], coins[2]), "Knuts")
```

Notice how a list called `coins` is created. We can pass each value in by indexing using `0`, `1`, and so on.

请注意名为 `coins` 的列表是如何创建的。我们可以通过使用 `0`、`1` 等进行索引来传入每个值。

- This is getting quite verbose. Wouldn't it be nice if we could simply pass the list of coins to our function?

这越来越冗长了。如果我们能简单地将硬币列表传递给我们的函数，那不是很好吗？

- To enable the possibility of passing the entire list, we can use unpacking. In the text editor window, code as follows:

为了实现传递整个列表的可能性，我们可以使用 unpacking。在文本编辑器窗口中，编写如下代码：

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


coins = [100, 50, 25]

print(total(*coins), "Knuts")
```

Notice how a `*` unpacks the sequence of the list of coins and passes in each of its individual elements to `total`.

请注意 `*` 如何解包硬币列表的序列，并将其每个单独的元素传递给 `total`。

- Suppose that we could pass in the names of the currency in any order? In the text editor window, code as follows:

假设我们可以按任何顺序传入货币的名称？在文本编辑器窗口中，编写如下代码：

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


print(total(galleons=100, sickles=50, knuts=25), "Knuts")
```

Notice how this still calculates correctly.

请注意这仍然是如何正确计算的。

- When you start talking about "names" and "values," dictionaries might start coming to mind! You can implement this as a dictionary. In the text editor window, code as follows:

当您开始谈论 "names" 和 "values" 时，词典可能会开始浮现在脑海中！您可以将其实现为字典。在文本编辑器窗口中，编写如下代码：

```
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")
```

Notice how a dictionary called `coins` is provided. We can index into it using keys, such as "galleons" or "sickles".

请注意如何提供名为 `coins` 的字典。我们可以使用键（例如 "galleons" 或 "sickles"）对它进行索引。

- Since the `total` function expects three arguments, we cannot pass in a dictionary. We can use unpacking to help with this. In the text editor window, code as follows:

由于 `total` 函数需要三个参数，因此我们不能传入字典。我们可以使用 unpacking 来帮助解决这个问题。在文本编辑器窗口中，编写如下代码：

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


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

print(total(**coins), "Knuts")
```

Notice how `**` allows you to unpack a dictionary. When unpacking a dictionary, it provides both the keys and values.

请注意 `**` 如何允许您解压缩字典。解压缩字典时，它会同时提供 keys 和 value。

`args` and `kwargs`

`args` 和 `kwargs`

- Recall the `print` documentation we looked at earlier in this course:

回想一下我们在本课程前面看到的 印刷 文档：

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

- `args` are positional arguments, such as those we provide to print like `print("Hello", "World")`.

`args` 是位置参数，例如我们提供给 print 的参数，如 `print("Hello", "World")`。

- `kwargs` are named arguments, or "keyword arguments", such as those we provide to print like `print(end="")`.
  `kwargs` 是命名参数，或"关键字参数"，例如我们提供给打印的那些，如 `print(end="")`。

- As we see in the prototype for the `print` function above, we can tell our function to expect a presently unknown number positional arguments. We can also tell it to expect a presently unknown number of keyword arguments. In the text editor window, code as follows:
  正如我们在上面的 `print` 函数原型中看到的那样，我们可以告诉我们的函数需要一个当前未知的数字位置参数。我们还可以告诉它期望当前未知数量的 keyword 参数。在文本编辑器窗口中，编写如下代码：

```
def f(*args, **kwargs):
    print("Positional:", args)


f(100, 50, 25)
```

Notice how executing this code will be printed as positional arguments.
请注意执行此代码将如何打印为位置参数。

- We can even pass in named arguments. In the text editor window, code as follows:
  我们甚至可以传入命名参数。在文本编辑器窗口中，编写如下代码：

```
def f(*args, **kwargs):
    print("Named:", kwargs)


f(galleons=100, sickles=50, knuts=25)
```

Notice how the named values are provided in the form of a dictionary.
请注意命名值是如何以字典的形式提供的。

- Thinking about the `print` function above, you can see how `*objects` takes any number of positional arguments.
  考虑上面的 `print` 函数，你可以看到 `*objects` 如何接受任意数量的位置参数。

- You can learn more in Python's documentation of `print`.
  您可以在 Python 的 `print` 文档中了解更多信息。

`map` 地图

- Early on, we began with procedural programming.
  早期，我们从过程式编程开始。

- We later revealed Python is an object oriented programming language.
  我们后来揭示了 Python 是一种面向对象的编程语言。

- We saw hints of functional programming, where functions have side effects without a return value. We can illustrate this in the text editor window, type `code yell.py` and code as follows:
  我们看到了函数式编程的提示，其中函数具有没有返回值的副作用。我们可以在文本编辑器窗口中对此进行说明，键入 `code yell.py` 和 code，如下所示：

```
def main():
    yell("This is CS50")


def yell(word):
    print(word.upper())


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

Notice how the `yell` function is simply yelled.
请注意 `yell` 函数是如何简单地被 yelled 的。

- Wouldn't it be nice to yell a list of unlimited words? Modify your code as follows:
  大喊一个无限单词的列表不是很好吗？按如下方式修改代码：

```
def main():
    yell(["This", "is", "CS50"])


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


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

Notice we accumulate the uppercase words, iterating over each of the words and uppercasing them. The uppercase list is printed utilizing the `*` to unpack it.
请注意，我们累积了大写单词，迭代每个单词并将它们大写。大写列表使用 `*` 进行解压缩。

- Removing the brackets, we can pass the words in as arguments. In the text editor window, code as follows:
  去掉括号，我们可以将单词作为参数传入。在文本编辑器窗口中，编写如下代码：

```
def main():
    yell("This", "is", "CS50")


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


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

Notice how `*words` allows for many arguments to be taken by the function.
请注意 `*words` 如何允许函数采用许多参数。

- `map` allows you to map a function to a sequence of values. In practice, we can code as follows:

`map` 允许您将函数映射到一系列值。在实践中，我们可以按如下方式编写代码：

```
def main():
    yell("This", "is", "CS50")


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


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

Notice how `map` takes two arguments. First, it takes a function we want applied to every element of a list. Second, it takes that list itself, to which we'll apply the aforementioned function. Hence, all words in `words` will be handed to the `str.upper` function and returned to `uppercased`.

注意 `map` 是如何接受两个参数的。首先，它需要一个我们想要应用于列表每个元素的函数。其次，它需要该列表本身，我们将应用上述函数。因此，`单词` 中的所有单词都将传递给 `str.upper` 函数并返回大写 。

- You can learn more in Python's documentation of `map`.

  您可以在 Python 的 `map` 文档中了解更多信息。

List Comprehensions 列表推导式

- List comprehensions allow you to create a list on the fly in one elegant one-liner.

  列表推导式允许您用一个优雅的单行代码动态创建一个列表。

- We can implement this in our code as follows:

  我们可以在代码中实现这一点，如下所示：

```
def main():
    yell("This", "is", "CS50")


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


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

Notice how instead of using `map`, we write a Python expression within square brackets. For each argument, `.upper` is applied to it.

请注意，我们不是使用 `map`，而是在方括号内编写 Python 表达式。对于每个参数，将对其应用 `.upper`。

- Taking this concept further, let's pivot toward another program.

  进一步了解这个概念，让我们转向另一个程序。

- In the text editor window, type `code gryffindors.py` and code as follows:

  在文本编辑器窗口中，键入 `code gryffindors.py` 和 code，如下所示：

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

gryffindors = []
for student in students:
    if student["house"] == "Gryffindor":
        gryffindors.append(student["name"])

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

Notice we have a conditional while we're creating our list. *If* the student's house is Gryffindor, we append the student to the list of names. Finally, we print all the names.

请注意，我们在创建列表时有一个条件。*如果*学生的学院是 Gryffindor，我们会将学生附加到姓名列表中。最后，我们打印所有名称。

- More elegantly, we can simplify this code with a list comprehension as follows:

  更优雅地，我们可以使用列表推导式来简化这段代码，如下所示：

```
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)
```

Notice how the list comprehension is on a single line!

请注意列表推导式是如何在一行上的！

`filter` 滤波器

- Using Python's `filter` function allows us to return a subset of a sequence for which a certain condition is true.

  使用 Python 的 `filter` 函数，我们可以返回特定条件为 true 的序列子集。

- In the text editor window, code as follows:

  在文本编辑器窗口中，编写如下代码：

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

```
def is_gryffindor(s):
    return s["house"] == "Gryffindor"


gryffindors = filter(is_gryffindor, students)

for gryffindor in sorted(gryffindors, key=lambda s: s["name"]):
    print(gryffindor["name"])
```

Notice how a function called `is_gryffindor` is created. This is our filtering function that will take a student `s`, and return `True` or `False` depending on whether the student's house is Gryffindor. You can see the new `filter` function takes two arguments. First, it takes the function that will be applied to each element in a sequence—in this case, `is_gryffindor`. Second, it takes the sequence to which it will apply the filtering function—in this case, `students`. In `gryffindors`, we should see only those students who are in Gryffindor.

请注意名为 `is_gryffindor` 的函数是如何创建的。这是我们的过滤函数，它将采用学生的 `s`，并根据学生的学院是否为格兰芬多学院返回 `True` 或 `False`。你可以看到新的 `filter` 函数接受两个参数。首先，它采用将应用于序列中每个元素的函数 – 在本例中为 `is_gryffindor`。其次，它采用将应用筛选函数的顺序 – 在本例中为 `students`。在 格兰芬多，我们应该只看到那些在格兰芬多的学生。

- `filter` can also use lambda functions as follows:

  筛选条件 还可以使用 Lambda 函数，如下所示：

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


gryffindors = filter(lambda s: s["house"] == "Gryffindor", students)

for gryffindor in sorted(gryffindors, key=lambda s: s["name"]):
    print(gryffindor["name"])
```

Notice how the same list of students is provided.

请注意，相同的学生列表是如何提供的。

- You can learn more in Python's documentation of `filter`.

  您可以在 Python 的 `filter` 文档中了解更多信息。

Dictionary Comprehensions

词典推导式

- We can apply the same idea behind list comprehensions to dictionaries. In the text editor window, code as follows:

  我们可以将列表推导式背后的相同想法应用于字典。在文本编辑器窗口中，编写如下代码：

```
students = ["Hermione", "Harry", "Ron"]

gryffindors = []

for student in students:
    gryffindors.append({"name": student, "house": "Gryffindor"})

print(gryffindors)
```

Notice how this code doesn't (yet!) use any comprehensions. Instead, it follows the same paradigms we have seen before.

请注意，这段代码（还）没有使用任何推导式。相反，它遵循我们以前看到的相同范式。

- We can now apply dictionary comprehensions by modifying our code as follows:

  现在，我们可以通过修改代码来应用字典推导式，如下所示：

```
students = ["Hermione", "Harry", "Ron"]

gryffindors = [{"name": student, "house": "Gryffindor"} for student in students]

print(gryffindors)
```

Notice how all the prior code is simplified into a single line where the structure of the dictionary is provided for each `student` in `students`.

请注意，所有前面的代码是如何简化为一行的，其中为 `students` 中的每个 学生 提供了字典的结构。

- We can even simplify further as follows:

  我们甚至可以进一步简化如下：

```
students = ["Hermione", "Harry", "Ron"]

gryffindors = {student: "Gryffindor" for student in students}

print(gryffindors)
```

Notice how the dictionary will be constructed with key-value pairs.

请注意如何使用键值对构建字典。

`enumerate` 枚举

- We may wish to provide some ranking of each student. In the text editor window, code as follows:

  我们可能希望提供每个学生的一些排名。在文本编辑器窗口中，编写如下代码：

```
students = ["Hermione", "Harry", "Ron"]

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

Notice how each student is enumerated when running this code.

请注意在运行此代码时如何枚举每个学生。

- Utilizing enumeration, we can do the same:

  利用枚举，我们可以做同样的事情：

```
students = ["Hermione", "Harry", "Ron"]
```

```
for i, student in enumerate(students):
    print(i + 1, student)
```

Notice how enumerate presents the index and the value of each `student`.

请注意 enumerate 如何显示每个 学生 的索引和值。

- You can learn more in Python's documentation of `enumerate`.

  您可以在 Python 的 `enumerate` 文档中了解更多信息。

Generators and Iterators 生成器和迭代器

- In Python, there is a way to protect against your system running out of resources the problems they are addressing become too large.

  在 Python 中，有一种方法可以防止您的系统耗尽资源，因为它们正在解决的问题变得太大。

- In the United States, it's customary to "count sheep" in one's mind when one is having a hard time falling asleep.

  在美国，当一个人难以入睡时，人们通常会在脑海中“数羊”。

- In the text editor window, type `code sleep.py` and code as follows:

  在文本编辑器窗口中，键入 `code sleep.py` 和 code，如下所示：

```
n = int(input("What's n? "))
for i in range(n):
    print("🐑" * i)
```

Notice how this program will count the number of sheep you ask of it.

请注意这个程序将如何计算您要求的绵羊数量。

- We can make our program more sophisticated by adding a `main` function by coding as follows:

  我们可以通过添加 `main` 函数来使我们的程序更加复杂，如下所示：

```
def main():
    n = int(input("What's n? "))
    for i in range(n):
        print("🐑" * i)


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

Notice how a `main` function is provided.

请注意 `main` 函数是如何提供的。

- We have been getting into the habit of abstracting away parts of our code.

  我们已经养成了抽象出部分代码的习惯。

- We can call a sheep function by modifying our code as follows:

  我们可以通过修改代码来调用 sheep 函数，如下所示：

```
def main():
    n = int(input("What's n? "))
    for i in range(n):
        print(sheep(i))


def sheep(n):
    return "🐑" * n


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

Notice how the `main` function does the iteration.

请注意 `main` 函数如何执行迭代。

- We can provide the `sheep` function more abilities. In the text editor window, code as follows:

  我们可以为 `sheep` 功能提供更多能力。在文本编辑器窗口中，编写如下代码：

```
def main():
    n = int(input("What's n? "))
    for s in sheep(n):
        print(s)


def sheep(n):
    flock = []
    for i in range(n):
        flock.append("🐑" * i)
    return flock


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

Notice how we create a flock of sheep and return the `flock`.

注意我们是如何创造一群羊并归还 羊 群的。

- Executing our code, you might try different numbers of sheep such as `10`, `1000`, and `10000`. What if you asked for `1000000` sheep, your program might completely hang or crash. Because you have attempted to generate a massive list of sheep, your computer may be struggling to complete the computation.

  执行我们的代码，您可以尝试不同数量的绵羊，例如 `10`、`1000` 和 `10000`。如果你要求 `1000000` 只羊，你的程序可能会完全挂起或崩溃。因为您尝试生成了大量的绵羊列表，所以您的计算机可能正在努力完成计算。

- The `yield` generator can solve this problem by returning a small bit of the results at a time. In the text editor window, code as follows:

  `yield` generator 可以通过一次返回一小部分结果来解决此问题。在文本编辑器窗口中，编写如下代码：

```
def main():
    n = int(input("What's n? "))
    for s in sheep(n):
        print(s)


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

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

Notice how `yield` provides only one value at a time while the `for` loop keeps working.

请注意 `yield` 如何一次只提供一个值，而 `for` 循环保持工作。

- You can learn more in Python's documentation of generators.

  您可以在 Python 的生成器文档中了解更多信息。

- You can learn more in Python's documentation of iterators.

  您可以在 Python 的迭代器文档中了解更多信息。

Congratulations! 祝贺！

- As you exit from this course, you have more of a mental model and toolbox to address programming-related problems.

  当您从本课程毕业时，您将有更多的心智模型和工具箱来解决与编程相关的问题。

- First, you learned about functions and variables.

  首先，您了解了函数和变量。

- Second, you learned about conditionals.

  其次，您了解了条件语句。

- Third, you learned about loops.

  第三，您了解了循环。

- Fourth, you learned about exceptions.

  第四，您了解了异常。

- Fifth, you learned about libraries.

  第五，您了解了库。

- Sixth, you learned about unit tests.

  第六，您学习了单元测试。

- Seventh, you learned about file I/O.

  第七，您了解了文件 I/O。

- Eighth, you learned about regular expressions.

  第八，您学习了正则表达式。

- Most recently, you learned about object-oriented programming.

  最近，您了解了面向对象的编程。

- Today, you learned about many other tools you can use.

  今天，您了解了许多其他可以使用的工具。

This was CS50! 这是 CS50!

- Creating a final program together, type `code say.py` in your terminal window and code as follows:

  一起创建最终程序，在终端窗口中键入 `code say.py` 并按如下方式编写代码：

```
import cowsay
import pyttsx3

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

Notice how running this program provides you with a spirited send-off.

请注意运行此程序如何为您提供热情洋溢的欢送。

- Our great hope is that you will use what you learned in this course to address real problems in the world, making our globe a better place.

  我们非常希望您将利用您在本课程中学到的知识来解决世界上的实际问题，使我们的地球变得更美好。

- This was CS50! 这是 CS50!