Python
Lecture 7

Modules

Chapter 1

Modules Python Introduction

In the world of programming, we care a lot about making code reusable. In most cases, we write code so that it can be reusable for ourselves. But sometimes we share code that's helpful across a broad range of situations.

In this lesson, we'll explore how to use tools other people have built in Python that are not included automatically for you when you install Python. Python allows us to package code into files or sets of files called *modules*.

A module is a collection of Python declarations intended broadly to be used as a tool. Modules are also often referred to as "libraries" or "packages" — a package is really a directory that holds a collection of modules.

Usually, to use a module in a file, the basic syntax you need at the top of that file is:

```
from module_name import object_name
```

Often, a library will include a lot of code that you don't need that may slow down your program or conflict with existing code. Because of this, it makes sense to only import what you need.

One common library that comes as part of the Python Standard Library is `datetime`. `datetime` helps you work with dates and times in Python.

Let's get started by importing and using the `datetime` module. In this case, you'll notice that `datetime` is both the name of the library *and* the name of the object that you are importing.

Example

```
from datetime import datetime

current_time = datetime.now()
print(current_time)
```

Modules Python Random

`datetime` is just the beginning. There are hundreds of Python modules that you can use. Another one of the most commonly used is `random` which allows you to generate numbers or select items at random.

With `random`, we'll be using more than one piece of the module's functionality, so the import syntax will look like:

```
import random
```
We'll work with two common `random` functions:

- `random.choice()` which takes a list as an argument and returns a number from the list
- `random.randint()` which takes two numbers as arguments and generates a random number between the two numbers you passed in

Let's take randomness to a whole new level by picking a random number from a list of randomly generated numbers between 1 and 100.

Example

```
import random

# Create random_list below:
random_list = [random.randint(1,100) for x in range(101)]
print(random_list)

# Create randomer_number below:
randomer_number = random.choice(random_list)

# Print randomer_number below:
print(randomer_number)
```

Modules Python Decimals

Let's say you are writing software that handles monetary transactions. If you used Python's built-in floating-point arithmetic to calculate a sum, it would result in a weirdly formatted number.

```
cost_of_gum = 0.10
cost_of_gumdrop = 0.35

cost_of_transaction = cost_of_gum + cost_of_gumdrop
# Returns 0.44999999999999996
```
Being familiar with rounding errors in floating-point arithmetic you want to use a data type that performs decimal arithmetic more accurately. You could do the following:

```
from decimal import Decimal

cost_of_gum = Decimal('0.10')
```

```
cost_of_gumdrop = Decimal('0.35')

cost_of_transaction = cost_of_gum + cost_of_gumdrop
# Returns 0.45 instead of 0.44999999999999996
```
Above, we use the `decimal` module's `Decimal` data type to add 0.10 with 0.35. Since we used the `Decimal` type the arithmetic acts much more as expected.

Usually, modules will provide functions or data types that we can then use to solve a general problem, allowing us more time to focus on the software that we are building to solve a more specific problem.

Ready, set, fix some floating point math by using decimals!

Example

```
from decimal import Decimal


# Fix the floating point math below:
two_decimal_points = Decimal('0.2') + Decimal('0.69')
print(two_decimal_points)


four_decimal_points =  Decimal('0.53') *  Decimal('0.65')
print(four_decimal_points)
```

Modules Python Files and Scope

You may remember the concept of *scope* from when you were learning about functions in Python. If a variable is defined *inside* of a function, it will not be accessible *outside* of the function.

Scope also applies to *classes* and to the *files* you are working within.

Files have scope? You may be wondering.

Yes. Even files inside the *same directory* do not have access to each other's variables, functions, classes, or any other code. So if I have a file **sandwiches.py** and another file **hungry_people.py**, how do I give my hungry people access to all the sandwiches I defined?

Well, *files are actually modules*, so you can give a file access to another file's content using that glorious `import` statement.

With a single line of `from sandwiches import sandwiches` at the top of **hungry_people.py**, the hungry people will have all the sandwiches they could ever want.