CS50's Introduction to Programming with Python

cs50.harvard.edu/python/2022/notes/9

Lecture 9

Et Cetera

- Over the many past lessons, we have covered so much related to Python!
- In this lesson, we will be focusing upon many of the "et cetera" items not previously discussed. "Et cetera" literally means "and the rest"!
- Indeed, if you look at the Python documentation, you will find quite "the rest" of other features.

`set`

- In math, a set would be considered a set of numbers without any duplicates.
- In the text editor window, code as follows:

```
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`.

- It turns out we can use the built-in `set` features to eliminate duplicates.

- In the text editor window, code as follows:

```
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.

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

Global Variables

- In other programming languages, there is the notion of global variables that are accessible to any function.
- We can leverage this ability within Python. In the text editor window, code as follows:

```
balance = 0


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


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

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

- 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:

```
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.

- 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:

```
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!

- 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:

```
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`.

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

Constants

- 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.

- In the text editor window, code as follows:

```
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!

- 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:

```
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`.

Type Hints

- In other programming languages, one expresses explicitly what variable type you want to use.
- As we saw earlier in the course, Python does require the explicit declaration of types.
- Nevertheless, it's good practice need to ensure all of your variables are of the right type.
- `mypy` is a program that can help you test to make sure all your variables are of the right type.
- You can install `mypy` by executing in your terminal window: `pip install mypy`.

In the text editor window, code as follows:

```
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`!

- 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:

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


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

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

- After installing `mypy`, execute `mypy meows.py` in the terminal window. `mypy` will provide some guidance about how to fix this error.
- You can annotate all your variables. In the text editor window, code as follows:

  ```
  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.

- Again, executing `mypy meows.py` in the terminal window provides much more specific feedback to you, the programmer.
- We can fix our final error by coding as follows:

  ```
  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.

- 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:

```
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`.

- 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:

```
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.

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

```
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.

- You can learn more in Python's documentation of Type Hints.
- You can learn more about `mypy` through the program's own documentation.

Docstrings

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

```
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.

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

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

`argparse`

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

```
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.

- 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!
- 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:

```
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.

- 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:

```
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.

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

```
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.

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

Unpacking

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

```
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.

- 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:

```
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.

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

```
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.

- This is getting quite verbose. Wouldn't it be nice if we could simply pass the list of coins to our function?
- To enable the possibility of passing the entire list, we can use unpacking. In the text editor window, code as follows:

```
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`.

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

```
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.

- 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:

```
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”.

- 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:

```
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.

`args` and `kwargs`

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

```
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")`.
- `kwargs` are named arguments, or “keyword arguments”, such as those we provide to print like `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:

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


f(100, 50, 25)
```

  Notice how executing this code will be printed as positional arguments.

- We can even pass in named arguments. In the text editor window, code as follows:

```
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.

- Thinking about the `print` function above, you can see how `*objects` takes any number of positional arguments.
- You can learn more in Python's documentation of `print`.

map

- Early on, we began with procedural programming.
- We later revealed Python is an object oriented programming language.
- 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:

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


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


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

  Notice how the `yell` function is simply yelled.

- Wouldn't it be nice to yell a list of unlimited words? Modify your code as follows:

```
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.

- Removing the brackets, we can pass the words in as arguments. In the text editor window, code as follows:

```
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.

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

```
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`.

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

List Comprehensions

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

- We can implement this in our code as follows:

```
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.

- Taking this concept further, let's pivot toward another program.
- In the text editor window, type `code gryffindors.py` and code as follows:

```
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.

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

```
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!

`filter`

- Using Python's `filter` function allows us to return a subset of a sequence for which a certain condition is true.
- In the text editor window, code as follows:

```
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.

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

```
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.

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

Dictionary Comprehensions

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

```
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.

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

```
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`.

- We can even simplify further as follows:

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

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

print(gryffindors)
```

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

`enumerate`

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

```
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.

- Utilizing enumeration, we can do the same:

```
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`.

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

Generators and Iterators

- In Python, there is a way to protect against your system running out of resources the problems they are addressing become too large.
- In the United States, it's customary to "count sheep" in one's mind when one is having a hard time falling asleep.
- In the text editor window, type `code sleep.py` and code as follows:

```
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.

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

```
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.

- We have been getting into the habit of abstracting away parts of our code.
- We can call a sheep function by modifying our code as follows:

```
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.

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

```
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`.

- 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.
- 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:

```
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.

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

Congratulations!

- As you exit from this course, you have more of a mental model and toolbox to address programming-related problems.
- First, you learned about functions and variables.
- Second, you learned about conditionals.
- Third, you learned about loops.
- Fourth, you learned about exceptions.
- Fifth, you learned about libraries.
- Sixth, you learned about unit tests.
- Seventh, you learned about file I/O.
- Eighth, you learned about regular expressions.
- Most recently, you learned about object-oriented programming.
- Today, you learned about many other tools you can use.

This was CS50!

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

```
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.

- 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.
- This was CS50!