Table of Contents

Lecture 8

Python Basics

- Today we'll learn to program in a new language, Python. The ability to learn new languages is important, but should be easier each time, as we build a foundation of patterns and concepts.
- We can map Scratch blocks to Python code as we did when we were first learning C.
- The `say` block in Python is just:

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

  - There's no more need for a `\n` or `;`, and `printf` is just `print` now.
- A main program that ran with a green flag in Python would be:

```
def main():
    print("hello, world")

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

  - Notice that here we can define a function in Python with a line like `def main():`, without any curly braces, but indentation instead to indicate the hierarchy. Notice that all lines nested under are indented with four spaces, and this needs to be precise and consistent for Python code to run correctly. We also use a colon, `:`, to indicate the start of a new block.
  - We don't need to specify the type of variable `main` will return.
- A loop to do something forever in Python looks like this:

```
while True:
    print("hello, world")
```

  - The boolean `True` has to be capitalized.
- A loop to do something a certain number of times could be achieved with this:

```
for i in range(50):
    print("hello, world")
```

    - The variable `i` is our placeholder again, but we don't need to declare it, and `range(50)` automatically creates a range of numbers (from `0` to `49` by default if we ask it for `50` numbers).
- Conditions are similar too:

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

    - Notice that we don't have parentheses around our expressions, anymore, and that instead of using curly braces, indentation is used to indicate which lines of code belong in which block.
    - And `elif` is the Python keyword that means `else if`.
- We'll also notice that while C was compiled, from source code to machine code, but Python is interpreted. This means that we can run our program with one command, which will then take care of everything that needs to be done to run the program.
- For example, we run a program we wrote with a command like `python hello.py`. This starts a program called `python`, which is then passed an argument, `hello.py`, that contains our program's source code.
- And underneath the hood, the `python` program is an interpreter that compiles our source code into something called bytecode, that can then run through our interpreter. But we can abstract away this process, and rely on the fact that we can simply use the interpreter to run our program.
- Python has familiar data types and features:
    - `bool`
    - `float`
    - `int`
    - `str`
    - `…`
- We've also implemented some training wheels again, with functions in a library that we'll call like this:
    - `get_char`
    - `get_float`
    - `get_int`
    - `get_string`
    - …
- And there are even more types and features built into Python, like:
    - `complex`
        - complex numbers from mathematics
    - `dict`
        - a dictionary, like a hash table
    - `list`
        - like an array that automatically grows and shrinks
    - `range`

    - `set`
        - a list with unique items, with operations like those of sets in mathematics
    - `tuple`
        - like structs, but without any specifications, like `(x, y)` to store two numbers
    - ...
- So let's save a file in the CS50 IDE, `hello.py`, with the following contents:

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

- Then, we can run `python hello.py` and see this:

```
$ python hello.py
hello, world
```

    - `python` is the name of the interpreter program that we've installed onto the CS50 IDE, and `hello.py` is the name of our file that we are passing in as an argument, for it to interpret.
- We can also do this:

```
from cs50 import get_string

s = get_string("name: ")
print(f"hello, {s}")
```

    - The syntax for including a library or a function is to use `import`, and we are importing `get_string` from the `cs50` library, which was pre-installed on the CS50 IDE.
    - Then we declare a variable called `s`, and not need to specify the type, and we call `get_string()` and store the return result into `s`.
    - Then we include `s` in what we print. Strings, or more generally objects, have built-in functions. We can call those functions with the syntax shown, like `f"hello, {s}"`, and by passing in the correct arguments, we can substitute variables the way we want. We also start the string oddly with an `f`, to indicate that it should be formatted.
- We can also use `print("hello, {}".format(s))` to indicate that we want to format a string. In Python, `"hello, {}"` is a string, `str`, which is actually an object that has built-in functions and features. `format` is one such function, which we can use to substitute variables into the string.
- We also wrote a program in C to get an integer from the user, and in Python `int.py` would look like:

```
from cs50 import get_int

def main():
    i = get_int("integer: ")
    print(f"hello, {i}")
```

    - But if we run this, nothing happens. We needed to add these lines to the end:

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

to call the function called `main`, which C calls for us automatically.

- We can write a familiar program that uses various operators:

```
from cs50 import get_int

# Prompt user for x
x = get_int("x: ")

# Prompt user for y
y = get_int("y: ")

# Perform arithmetic
print(f"{x} plus {x} is {x + y}")
print(f"{x} minus {y} is {x - y}")
print(f"{x} times {y} is {x * y}")
print(f"{x} truly divided by {y} is {x / y}")
print(f"{x} floor-divided by {y} is {x // y}")
print(f"remainder of {x} divided by {y} is {x % y}")
```

  - There is a special operator in Python, `//`, that divides two integers and returns an integer that's truncated (with everything after the decimal point removed). Otherwise, the `/` symbol will divide two integers into a float if needed.
  - And comments in Python, instead of starting with `//`, will start with `#`.
- We can add logic, too:

```
from cs50 import get_int

x = get_int("x: ")

y = get_int("y: ")

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

  - We get two integers and compare them. And in Python, single quotes `'` and double quotes `"` can both be used to indicate strings, as long as we use the same one on both sides of the string.
  - And notice that here we didn't define or call a `main` function, but it still runs top to bottom as a program. We'll be able to run it in our terminal, but we won't be able to import it in other programs.
- We can look at strings:

```
# Logical operators

from cs50 import get_char

# Prompt user for answer
c = get_char("answer: ")

# Check answer
if c == "Y" or c == "y":
    print("yes")
elif c == "N" or c == "n":
    print("no")
```

  - We call `get_char`, and compare it to `Y` or `y` or `N` or `n` to tell us if we said yes or no.
  - We just say `or` and `and` in Python instead of `||` and `&&`.
  - And in C, we needed to compare `char`s by using single quotes, but in Python single characters are also strings. The good news is, we can compare strings with a simple `==` and it will compare them the way we might expect, equalling `True` if the strings have the same contents.
- We can also define functions that return some values:

```
# Return value

from cs50 import get_int


def main():
    x = get_int("x: ")
    print(square(x))


def square(n):
    """Return square of n"""
    return n**2


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

  - We don't need to specify the return type of `square` when we declare it, or the type of arguments it needs.
  - The three double quotes in a row that start and end a string are indicating a comment, and we use three double quotes to write a docstring comment, which describes the function so we can automatically create documentation for our program.
- We can also write a program to get a positive integer from the user:

```
from cs50 import get_int

def main():
    i = get_positive_int("positive integer, please: ")
```

```
    print(i)

def get_positive_int(prompt):
    """Prompt user for positive integer"""
    while True:
        n = get_int(prompt)
        if n >= 1:
            break
    return n

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

  - Here, we didn't need to define `get_positive_int` before we called it, since it wasn't actually run before we got to the part of the code that defines it. In this case, we call `get_positive_int` in `main`, but `main` itself isn't called until the very last line, and by then everything in our program had already been defined.
  - Python also doesn't have a `do while` loop, so instead we use `while True`, and `break`, or stop the loop, `if n > 0`.
  - Then it returns `n`, but notice that we also didn't need to declare it outside the loop before we used it. `n` will be created the first time our loop runs, and then have the new value stored inside it every time after.
  - And finally, we need to call the `main` function with the last two lines.
- In lecture, David wrote the following, which actually won't run, because `get_positive_int` wasn't defined when it was actually called:

```
from cs50 import get_int

i = get_positive_int("positive integer, please: ")
print(i)

def get_positive_int(prompt):
    """Prompt user for positive integer"""
    while True:
        n = get_int(prompt)
        if n >= 1:
            break
    return n
```

More Python

- We can observe integer overflow in Python as well:

```
from time import sleep

# Iteratively double i
i = 1
while True:
    print(i)
    i *= 2
    sleep(1)
```

    - If we run this, we see larger and larger values of `i`. We needed to import the `time` library for our program to pause for one second each time it prints a number.
    - But the Python maximum for an integer is far larger than the maximum it is in C.
- Let's revisit our favorite friend, Mario:

```
# Prints four question marks

print("????")
```

- We can write that same program with a loop:

```
# Prints four question marks using a loop

for i in range(4):
    print("?", end="")
print()
```

    - `print` also seems to take other arguments, which we can name, and here we are passing in an argument for `end`, where we specify that the ending is an empty string (as opposed to the default, a new line).
    - For functions with multiple optional arguments, using this method to name arguments as we pass them in, will mean that we can pass any of them in any order.
- We can combine concepts and print any number of question marks in a loop:

```
# Prints any number of question marks, as specified by user

from cs50 import get_int

n = get_int("Number: ")
for i in range(n):
    print("?", end="")
print()
```

- And we can check that the number is positive:

```
# Prints a positive number of question marks, as specified by user

from cs50 import get_int

# Prompt user for a positive number
while True:
    n = get_int("Positive number: ")
    if n > 0:
        break

# Print out that many bricks
for i in range(n):
    print("#")
```

- And lastly, we can print a square of comments with nested loops:

```
# Prints a square of bricks, sized as specified by user

from cs50 import get_int

# Prompt user for a positive number
while True:
    n = get_int("Positive number: ")
    if n > 0:
        break

# Print out this many rows
for i in range(n):

    # Print out this many columns
    for j in range(n):
        print("#", end="")
    print()
```

  - `print()` gives us a new line automatically, and we add it to the end of the outer loop.
- We can use command-line arguments too:

```
import sys

if len(sys.argv) == 2:
    print(f"hello, {sys.argv[1]}")
```

  - We can check the length of the arguments with `len(sys.argv)`, and access the second one (recall that the first is our program's own name) with `sys.argv[1]`. Here `sys` is a module built into Python that has command-line arguments and others.
- We can print all of the arguments too:

```
import sys

for s in sys.argv:
    print(s)
```

- And we can print each character in each argument:

```
import sys

for s in sys.argv:
    for c in s:
        print(c)
    print()
```

  - With `for s in sys.argv`, we are accessing element in `sys.argv`, and calling it `s`. And the type of each element will be a string.
  - Then with `for c in s`, we are accessing each element in the string `s`, which we will call `c`, since each element is a character.
- We can get the initials from a string passed in:

```
# Extracts a user's initials

from cs50 import get_string

s = get_string("Name: ")
initials = ""
for c in s:
    if c.isupper():
        initials += c
print(initials)
```

  - Here, we iterate over the characters in `s`, and if they are uppercase, append, or add, it to the string `initials` that we initialized as an empty string. In Python, all we need is `+=` to add to a string.
  - And notice that `c`, even though it's only a single character, is still a string in Python, so we are able to use the same built-in functions that every string in Python comes with.
- We can search in a list with just one line:

```
# Linear search

import sys
from cs50 import get_string

# Names in a phone book
book = [
    "Chen",
    "Kernighan",
    "Leitner",
```

```
    "Lewis",
    "Malan",
    "Muller",
    "Seltzer",
    "Shieber",
    "Smith"]

# Prompt user for name
name = get_string("Name: ");

# Search for name
if name in book:
    print(f"Calling {name}")
    sys.exit(0)
print("Quitting")
```

  - We declare a list of strings named `book`.
  - All we neede is `if name in book`, and the search happens for us automatically.
- We can compare two strings the way we expected:

```
from cs50 import get_string

# Get two strings
s = get_string("s: ")
t = get_string("t: ")

# Compare strings for equality
if s == t:
    print("same")
else:
    print("different")
```

  - Instead of `null`, there is a special value that `get_string` might return, `None`, that indicates there is nothing returned.
- Since we don't have access to pointers in Python, we aren't able to swap the values of two variables by passing their pointers to a function. Instead, we can simply do this:

```
x = 1
y = 2

print(f"x is {x}, y is {y}")
x, y = y, x
print(f"x is {x}, y is {y}")
```

  - The left side and right side, `x, y`, and `y, x` are both tuples, a data structure with multiple values, and we're setting the items inside `x, y` to what the items inside `y, x` are, which swaps the values.
- Let's implement structures in Python, which are called classes:

```
```

```
from cs50 import get_string
from student import Student

# Space for students
students = []

# Prompt for students' names and dorms
for i in range(3):
    name = get_string("name: ")
    dorm = get_string("dorm: ")
    students.append(Student(name, dorm))

# Print students' names and dorms
for student in students:
    print(f"{student.name} is in {student.dorm}.")
```

  - First, we declare a `student` file that we'll soon write, and import the `Student` class from it.
  - Then we can create an empty list to store students called `students`, which we can add or remove things to.
  - Then we get a `name` and `dorm`, create a `Student` object by passing those strings in as arguments, and `append` it, or add it, to the end of our list `students`. (Lists, too, have built-in functionality, one of which is `append`.)
  - Finally, for each `student`, we print the properties back with the `.` syntax.
- So to create our `student` module, we would:

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

  - We declare a `class` of objects called `Student`, which will only have one method, or built-in function, `*_init_*`, which we won't call directly but gets called when we create a `Student` as we did above with `Student(name, dorm)`.
  - This function gets the object itself as an argument and the other arguments we want to be passed in when the object is created, in this case `name` and `dorm`. Then inside the function, we store the arguments to the object that's just been created.
- We can see another convenient feature, storing our `students` to a file in `struct1.py`:

```
# Demonstrates file I/O

import csv
from cs50 import get_string
from student import Student

# Space for students
students = []

# Prompt for students' names and dorms
for i in range(3):
    name = get_string("name: ")
```

```
    dorm = get_string("dorm: ")
    students.append(Student(name, dorm))

with open("students.csv", "w") as file:
    writer = csv.writer(file)
    for student in students:
        writer.writerow((student.name, student.dorm))
```

  - Now, instead of printing the students to the screen, we can write them to a file `students.csv` by opening it and using a built-in module, `csv`, that writes comma-separated values to files.
  - With `csv.writer(file)`, we pass in the file we open to get back a `writer` object that will take in tuples, and write them to the file for us with just `writerow`.
- We can re-implement all the examples from weeks 1 through 5 in Python, and even the entire `speller` program.
- More interestingly, we can look at just the `dictionary.py` file:

```
class Dictionary:
    """Implements a dictionary's functionality"""

    def __init__(self):
        self.words = set()

    def check(self, word):
        """Return true if word is in dictionary else false"""
        return word.lower() in self.words

    def load(self, dictionary):
        """Load dictionary into memory, returning true if successful else false"""
        file = open(dictionary, "r")
        for line in file:
            self.words.add(line.rstrip("\n"))
        file.close()
        return True

    def size(self):
        """Returns number of words in dictionary if loaded else 0 if not yet
loaded"""
        return len(self.words)

    def unload(self):
        """Unloads dictionary from memory, returning true if successful else false"""
        return True
```

  - Here, we create a `words` property when each `Dictionary` is initialized, and set it to an empty `set`. In Python, sets are abstracted away (so we don't know anything about how it's implemented in memory anymore, or whether it's a hash table, or trie, or something else entirely) but we can easily operate with it. This is an important tradeoff compared to C, where we were able to control implementation details and perhaps better tune performance for the needs of our program, at the cost of our own human time to develop it.

    - We can add items to `self.words` with `self.words.add()`, check if a word is in it with `word in self.words()`, and get the size with `len(self.words)`.
    - And since Python mananges the memory for us, we don't even need to worry about unloading it or freeing it.
- Ultimately, a higher-level language like Python, which has implemented many lower-level features that would take dozens of lines in C, allows us to write more and more sophisticated programs without having to worry about all of the details.