Questions of the day

- What do loops in Python look like?
- Does Python have an analogue to an array (or `ArrayList`)?
- How do we read and write files?

Loops, lists, and files

William Hendrix

Lecture 2

Outline

- Review
  - Variables
  - Branching
  - Functions
- Strings and f-strings
- For loops
- While loops
- Loop control
- File reading
- File writing

Review: Python basics

- Variables and assignments similar to Java
  - No `;` or data types
  - `# comments` or `'''docstrings'''`
- `print()`: basic output
  - Multiple arguments, `end='', sep=''`
- `input(prompt)`: basic input
  - Returns string by default
  - Typecast output with `int(input(...))` or `float(...)`

```
if condition:
    if_body
elif alternative:
    elif_body
else:
    else_body
```

  - Good indentation not optional!

Function review

Calling

```
function_name(arg1, arg2, ...)
```

– No args: `function_name()`

- **Examples**

```
print('Enter', name, ': ')
x = int(input())
```

Defining

```
def function_name(arg1, arg2, ...):
    function_body
```

– `return value`: ends function and outputs `value`

- **Example**

```
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Default and keyword arguments

- Keyword arguments

```
def divide(numerator, denominator):
    return numerator / denominator
# ...
print(divide(denominator = 14, numerator = 7)) #0.5
```

- **Syntax:** `arg_name = value`
  - Must appear *after* any positional arguments

```
print('Enter', name, ': ', end='')
```

- Default arguments

```
def get_tax(amt, rate = 0.06625):
```

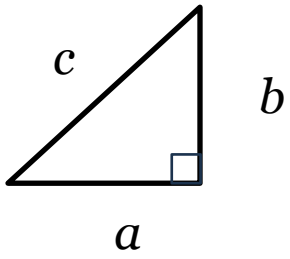
  - Must appear after args with no default
  - Argument becomes optional (`get_tax(5)` vs `get_tax(5, .07)`)
  - **Example:** `print(..., sep = ' ', end = '\n'))`

Function example

- I'll write a pair of functions, `find_left_root` and `find_right_root`, to find the two roots of a quadratic function
  - Arguments: `a, b, c`
    - Represents $y = ax^2 + bx + c$
  - Return:
    - Left root: $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$
    - Right root: $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$
  - *Hint:* exponentiation supports decimals
  - No special error handling if $b^2 - 4ac$ is negative
- **Examples:**
  - $x^2 - 2$: ±1.414213562373095 1
  - $2x^2 - 10x - 10$: -0.8541019662496847 and 5.854101966249685

Function exercise

- Write a `hypotenuse` function that accepts the length of two sides of a right triangle (*a* and *b*) and computes the length of the hypotenuse



  - Pythagorean Theorem: $a^2 + b^2 = c^2$
- **Examples**
  - 1, 1: 1.4142135623730951
  - 3, 4: 5.0
  - 7, 12: 13.0

Format strings

- Strings that print values in special ways
- **Syntax:** put `f` in front of string and `{...}` inside
  - Can have multiple `{}` in one f-string
- **Example:**

```
print(f'x = {x}')
print(f'10x^2 = {10 * x ** 2}')
```

- 1-3 things inside {}:
  - Variable or expression to print (required!)
  - Type conversion (optional)
    - E.g., `!r` or `!a`
    - Very rare
  - Format specification (optional)
    - Several useful options
    - E.g., `:^8` or `:.2`

Format specification

- Most common formatting option: rounding
- Specifies max number of decimal digits
  - `:.2f`: 2 decimal digits (money!)
  - `:.0f`: no decimal digits
  - Works with any number, but 0 and 2 are most common
- **Examples:**

```
print(f'Your total is ${total:.2f}')
print(f'Grade {grade:.0f}')
print(f'Pi to 4 places:  {pi:.4f}')
```

- Can specify field width and alignment
  - **Alignment:** left (<), right (>), or centered (^)
  - **Field width:** total number of spaces this value should occupy
  - Useful for printing tables
- Many, many more options (see documentation)

Sequences and collections

- Strings are different than int or float
  - Example of a *sequence*
  - Variable that stores several pieces of information in order

```
str = 'hello'
# 5 different characters:  h, e, l, l, then o
```

- *Collection*: general term for variables that store multiple pieces of information
  - Some collections are *unordered*

- Strings have several features numbers don't

Length and indexing

- `len(str)`: length of string `str`

- `str[i]`: print or test individual characters in a string
  - Put `[]` after variable name
  - `str[0]`: first character
  - `str[1]`: second character
  - …
  - `str[len(str) - 1]`: last character
  - Extracting characters is called *indexing*
    - *Index*: location of a character in a string
- Careful: if too large, causes IndexError
- Negative indexes count from end
  - `str[-1]`: last character
  - `str[-len(str)]`: first character

Loops

- Python supports two kinds of loops
  - `for` loops
  - `while` loops
- **`for` loop syntax**

```
for loop_variable in range(count):
    loop_body
```

  - Used when you know # iterations
  - Indentation rules same as `if..elif..else`
  - `loop_variable` becomes `0, 1, ..., count - 1`
- **Examples:**

```
for x in range(5):
  print('hi', end='') # hihihihihi
for x in range(3):
  print(f'#{x + 1}', end='') # #1#2#3
```

Changing the range

- `range()` can be used in two more ways
- `for i in range(start, end):`
  - All numbers from `start` up to `end - 1`
    - `end - start` iterations
- **Examples**

```
for i in range(10, 15):
    # 5 iterations:  10, 11, 12, 13, 14
for i in range(1, 101):  # 1-100
```

- `for i in range(start, end, increment):`
  - All numbers from `start` up to `end`, going by `increment`
  - `floor((end - start) / increment)` iterations
- **Example**

```
for i in range(0, 100, 2):
  # All even numbers from 0 to 98
```

`while` loops

- Repeats until a given condition fails
  - Used when # iterations not known
- **Syntax:**

```
while condition:
    # code to repeat...
```

  - Basically identical to `if` syntax
  - If `condition` is true, code gets executed
  - Condition is checked again after the loop and repeats until `False`
  - Make sure you're updating loop variable!
- **Example:**

```
num = int(input("Enter a number 1-100: "))
while num < 1 or num > 100:
    num = int(input("Bad input.  Enter 1-100: "))
```

Loop control

- Python supports two loop control statements
  - `break`: stops loop immediately
  - `continue`: skips to start of next iteration
- **Example:**

```
is_prime = True
m = 2
while m * m <= n:
    if n % m == 0:
        is_prime = False
        break
    m += 1
```

- Always possible to write loop without `break` or `continue`
  - **E.g.,** `while m * m <= n and is_prime:`

Loop `else`

- `for` and `while` loops can be followed by `else`
- `else` body executes unless `break` was used
- **Example:**

```
for k in range(2, n):
    if n % k == 0:
        print(f"{k} is a factor of {n}")
        break
else:
    print(f"{n} is prime")
```

Loop example

- I will write a script to prompt the user for two integers, `m` and `n`, and print a table of the function f(x) = x^2 for all values from `m` up to (but not including) `n`

```
Enter m:  9
Enter n:  15
   x     x^2
   9      81
  10     100
  11     121
  12     144
  13     169
  14     196
```

Loop exercise

- Write a script that prompts user for initial amount of money and number of bills
- Prompt user for amount of each bill
  - Subtract bill amount from money
  - Stop early if money becomes negative
- Print "`All bills paid`" if they were all paid

File I/O

- I/O: input/output
  - File I/O: reading and writing files
- `open(file_name)`: function that opens a file
  - Allows you to read existing file by default
    - Specify absolute or relative path if not in same directory
  - `open(file, mode='w')`: create or overwrite file
- **Example**

```
my_file = open('babynames.txt')
print('Popular baby names:')
contents = my_file.read()
print(contents)
my_file.close()
```

  - `file.read()`: returns string with file contents
  - `file.close()`: closes file

Reading a file

- 4 common ways to read a file:
  - `for line in file:`
    - Iterates through every line in the file
  - `file.readline()`: returns one line of the file
    - Each subsequent call gets the next line
    - Returns empty string (`''`) at end of file
  - `file.readlines()`: returns a *list* of strings with each line
  - `file.read()`: returns a single string with entire contents
- All have `\n` at end of each line
  - `str.rstrip()`: remove whitespace at end (`' ', '\n', '\t'`)
  - Or you can print with `end=''`

File writing overview

1. Open the file

```
file_var = open('filename', 'w')
```

   - Second argument is the mode
   - `'w'`: writing a file
     - Creates the file if it doesn't exist
     - Overwrites existing file otherwise
   - `'r'`: reading a file
     - Default mode
     - Fails if the file doesn't exist
   - `'a'`: appending to a file
     - Writes to the end of a file
     - Creates file if doesn't exist

2. Write the data

```
file_var.write('stuff to write\n')
```

   - Need `\n` at end if you want to end a line

3. Close the file

```
file_var.close() # Very important!
```

File example

- I'll write a script that reads and prints the top 20 girl and boy baby names
  - `babynames.txt`: top 1000 girl names from 2024, then top 1000 boy names

File writing exercise

- Write a program that prompts the user for a file name, then it prompts the user to write as much text as they like (including multiple lines), which will be written to the given file.
- When they type a blank line (enters twice in a row), the program will end and close the file.
- The file should contain everything the user typed except for the blank line at the end

`with` block

- **Common file error:** forgetting to `.close()` file
- `with` keyword
  - Opens file at start
  - Indent code related to file
  - Closes file automatically at end of `with` block
- **Syntax:**

```
with open('filename') as file:
    # Indent code to process file
```

- Equivalent to:

```
file = open('filename')
# ...code...
file.close()
```

- Can also be used with `open('filename', 'w')`

Lists

- Sequence that can store any kind of value
- **Syntax:** comma-separated list in brackets

```
my_list = [1, 2, 3, 4, 5]
```

- Lists can contain any kind of data

```
list1 = [1, 2, 3]
list2 = [1.414, 2.718, 3.14]
list3 = ['hello', 'world']
list4 = [[1, 2], [3, 4]]
```

- Can even store different data types

```
mixed_list = [1, '234', [5, 6]]
```

  - Usually a bad idea

- Mutable (unlike strings)

```
list1[1] = 17 # [1, 17, 3]
```

Common features with strings

- Concatenated like strings

```
my_list = my_list + [7, 8]
```

- Compared like strings
  - Compares first difference
    - Error if they are not the same type

```
[1, 2] >= [0, 1, 2]  # True (1 >= 0)
['a', 'b', 'c'] < ['a', 'bc'] # True ('b' < 'bc')
```

- Convert other collections with `list()`

```
list('hello') # ['h', 'e', 'l', 'l', 'o']
list(range(3)) # [0, 1, 2]
```

Useful list methods

- `my_list.append(x)`
  - Adds `x` to the end of `my_list`
  - Can be used in a loop to create list

- `my_list.pop(0)`
  - Remove element at given index from `my_list`
  - `.pop()`: removes last element

- `my_list.remove(5)`
  - Removes 5 from `my_list`
  - Removes first 5 if there are multiple
  - Error if 5 (or whatever) isn't there
  - `if 5 in my_list:`
    - `True` if contained

List iteration

- `for` loops can also be used with lists

```
for value in [1, 'abc', [1, 2]]:
    print(value)
my_list = [1, 2, 3]
for value in my_list:
    print(value)
```

  - Loop variable becomes first entry, then second, etc.
  - Similar to:

```
for i in range(len(my_list)):
    print(my_list[i])
```

- Also works with strings or any collection

```
for char in my_str:
    print(char)
```

Dictionaries

- Powerful data structure
  - Called a "map" in other languages
  - `HashMap` in Java
- **Syntax:**

```
my_dict = {key1 : value1, key2 : value2, ...}
```

- Look up, add, or change values using `dict_name[key]`
- **Example:**

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
lettercount = {} # Empty dictionary
for c in alphabet:
    lettercount[c] = 0
for c in text.lower():
    if c in alphabet:
        lettercount[c] = lettercount[c] + 1
        # or:  lettercount[c] += 1
```

Common features with lists

- Test whether a given key is defined using `in`:

```
test = {0 : 'zero', 2 : 'two'}
if 0 in test: # True
    print('Success')
if 1 in test: # False
    print('Failure')
```

  - Only checks keys, not values

```
if 'zero' in test: # False
```

- Can iterate through keys or (key, value) pairs

```
for key in test:
    print(f'{key} : {test[key]}')
for key, value in test.items():
    print(f'{key} : {value}')
```

- `test.pop(key)`: removes corresponding entry

Coming up

- String processing
- Tuples
- Sets
- Recursion

- **Homework 2** due next Wednesday (Sept 18)
- **Recommended reading:** week 3 in textbook