# Introduction to Programming

| Status | Completed |
|---|---|

## ▼ Lecture 0: Functions, Variables

### 11/01/23

- Part of good programming is finding a balance between compactness and readability.
- Python is a program (also known as an interpreter) that you install on your computer. You can run it in the command line with:

```
$ python program.py
```

- To comment out one line in Python, use the `#` symbol:

```
# This is a comment
```

- For docstrings, use `“””`:

```
"""
    This is a docstring
"""
```

- The `print()` function prints to the command line.

```
# Print a string
print("Print this to the command line")

# Concatinate with '+'
print("Print this to the " + "command line")

# Concatinate with ','
print("Print this to the", "command line")

# Documentation
    # sep= and end= are named parameters
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Parameters vs. arguments
  - Parameters are defined with a function, where arguments are what one actually passes into a function.
- To indicate a new line use `\n`:

```
print("First line.\n", "Second line")
```

- To use quotes use `\”`:

```
print("Hello, \"friend\".")
```

- f-strings allow you to insert variable values into strings easily:

```
print(f"The value of x is: {x}.")
```

- `.strip()` can be used to remove characters from a string:

```
# Remove white space from a string
string = string.strip()
```

- `.capitalize()` and .title() can be used to capitalize a string:

```
# Capitalize the first character of a string
string = string.capitalize()

# Capitalize the first character of each word in a string
string = string.title()
```

- .`split()` can split a string into two smaller substrings

```
# Split a two word string on the space
first, second = string.split(" ")
```

- Methods can be chained together:

```
# Remove whitespace from string and capitalize string
string = string.strip().title()
```

- Python can manipulate numbers:

```
# Add
x + y
# Subtract
x - y
# Multiply
x * y
# Divide
x / y
# Exponent
x ** y
# Modulo
x % y
```

- Python’s interactive mode allows you to execute commands directly in the command line:

```
# 5 will be output in the command line.
$ 10 / 2
```

- `round()` can be used to round a float to a certain decimal place:

```
# Round a float to 4 decimal places
x = round(x, 4)

# Documentation
    # [] indicate that the ndigits parameter is optional
round(number[, ndigits])
```

- `:,` can be used to add commas to a number

```
# if x = 10000, print 10,000
print(f"{x:,}")
```

---

### 11/02/23

- Python allows one to create custom functions:

```
# Create a custom function
def function_name():
    # Function contents are indented
```

- Functions can be given default parameters

```
# A function with a default parameter
def function_name(parameter_name="default):
```

- One of the points of defining your own functions is to avoid rewriting unnecessary code.
- Functions need to be defined above where it is used.
    - To get around this, define a main() function and call main() at the bottom of the file

```
# Main function
def main():
    func1()
    func2()
    # ...
    funcN()

# Other functions
def func1():
    # Function 1 contents
def func2():
    # Function 2 contents
def funcN():
    # Function n contents

# Call main after every function has been defined
main()
```

- To use a variable, it must be used within its scope.
- A function can return a value:

```
# This function returns a value
def func():
    return value
```

### Definitions

- Functions allow you to do “something” in the program.
- Parameters are what inputs a function is defined with.
- Arguments are inputs to a function when that function is used.
- Bugs are mistakes in a program.
- Variables store values.
- Comments are non-compiled code that allow code documentation.
- Pseudocode is fake code used to outline a program before writing it.
- Datatypes are the different “types” of variables that can be stored.
    - Strings are variables that store multiple characters of text.
    - Integers are variables that store whole numbers.
    - Floats are variables that store numbers with decimal points.
- Methods are functions built into a datatype.
- Operators perform operations on variables.
- Operands are the values that operators apply an action to.
- Scope refers to a variable only existing within the context in which one defines it.
- Return values ‘return’ variables at the end of functions.

## ▼ Lecture 1: Conditionals

### 11/03/23

- Conditionals allow logical forks to be taken in ones code.
    - There are many types of conditionals:

```
# Greater than
>
# Greater than or equal to
>=
# Less than
<
# Less than or equal to
<=
# Equal to
==
# Not equal to
!=
```

- Use `if` statements to tell Python to execute code if a condition is met:

```
# Conditional statement
if x == y:
    # Execute if statement code
```

- `else if` statements can take into account if a previous conditional was met. In other words, the two conditionals are mutually exclusive:

```
# If statement
if x > y:
    # Execute if statement code
elif x < y:
    # Execute else if statement code
```

- `else` statements can be run when no other conditionals in a chain are met:

```
# If statement
if x > y:
    # Execute if statement code
elif x < y:
    # Execute else if statement code
else:
    # Execute else statement code
```

- `or` is a Boolean operation that can check if either statement is true:

```
# If statement with an 'or' operator
if x < y or x > y:
    # Execute if statement code
```

- `and` is a Boolean operation that can check if both statements are true:

```
# If statement with an 'and' operator
if x > y and x < z:
    # Execute if statement code
```

- Conditionals can be chained together in a single statement:

```
# If statement with 2 conditionals
if y < x < z:
    # Execute if statement code
```

- A Boolean datatype can be true or false.
    - This is equivalent to 1 or 0:

```
# True
True == 1
# False
False == 0
```

- In Python, an if-else statement can be condensed:

```
# Before
if x == y:
    return True
else:
    return False

# After
return True if x == y else False
```

- Match statements are another syntax for conditionals:
    - `case _` can be used in place of else.

```
match var:
    case x:
        # Execute code
    case y:
        # Execute code
    case z:
        # Execute code
    case _
        # Execute code
```

- `or` is expressed slightly differently:

```
# Checking for a case of x 'or' y
match var:
    case x | y:
        # Execute code
```

### Definitions

- Conditionals allow logical forks to be taken in code.
- Boolean values are datatypes that are either true or false.
- Parity refers to if a number is even or odd in mathematics.
- Pythonic refers to the way some things are done in Python.
- Match statements express conditionals more compactly in certain cases.

## ▼ Lecture 2: Loops

### 11/07/23

- Loops allow one to repeat a line of code multiple times.
  - Loops save time, as less lines of code need to be edited if changes are needed.
- A `while` loop runs code while a condition is true:

```
# Run a line of code while 'i' is less than 3
i = 0
while i < 3:
    # Execute code in while loop
    i = i + 1
```

- ctrl + c can be used to cancel running ones code.
  - This is useful for if a while loop gets stuck running infinitely.
- Incrementations or decrementations can be compacted with syntactic sugar:

```
# Increment 'i' by 'n'
i = i + n
# Increment 'i' by 'n' with syntactic sugar
i += n

# Decrement 'i' by 'n'
i = i - n
# Decrement 'i' by 'n' with syntactic sugar
i -= n
```

- For loops allow one to iterate over a list:

```
# Run code 'n' times
for i in [0, 1, 2, n - 1]:
    # Execute code in loop
```

- This syntax can be compacted.
  - `range(n)` runs code from [0, i-1] inclusive.

```
# Run code 'n' times
for i in range(n):
    # Execute code in loop
```

- If the variable in the for loop isn't being used, it is Pythonic to make the variable an `_`:

```
# Run code 'n' times
for _ in range(n):
    # Execute code in loop
```

- `continue` and `break` can be used to either continue a while loop, or break out of a while loop:

```
# Run loop until n is greater than 0
while True:
    n = int(input("What's n? ))
    if n < 0:
        # Continue the loop
        continue
    else:
        # Break out of loop and execute the next line after the while loop
        break
```

- A list stores multiple values of the same datatype:

```
# Store the names of students in a list
names = ["Alice", "Bob", "Charlie", "Dale"]
```

- One can get the nth value from a list:

```
# Print "Bob"
print(names[1])
```

- For loops can be used to iterate over strings:

```
# Print all names
for name in names:
    print(name)
```

- The function `len()` returns the length of a list:

```
# Get amount of names
number_names = len(names)
```

- Dictionaries allow one to store key-value pairs in a list:

```
# Store students and their favorite subject in a dictionary
students = {
    "Alice": "Physics",
    "Bob": "Math",
    "Charlie": "History"
}
```

- Values are accessible by using the key:

```
# Print Alice's favorite subject
print(students["Alice"])

# Iterate over the dictionary and print all students' favorite subject
for student in students:
    print(students[student])
```

- A list of dictionaries can be created to store more information:

```
# Store multiple values per student
students = [
    {"name": "Alice", "subj": "Physics", "grade": 10},
    {"name": "Bob", "subj": "Math", "grade": 12},
    {"name": "Charlie", "subj": "History", "grade": 9},
]

# Iterate over the dictionary and print all students' information
for student in students:
    print(student["name"], student["subj"], student["grade"], sep=", ")
```

- `None` is used to represent the absence of a value:

```
# The variable has no value
var = None
```

## Definitions

- Loops allow code to run cyclically.
- While loops allow code to run while a condition is true.
- Syntactic sugar is code that does the same job as other code, but is either more compact, easier to understand, or easier to write.

- For loops are an iterative loop that run code over a list of items.
- Lists are a datatype that hold multiple values of the same datatype.
- Breaks exit out of the local code scope.
- Indices are ordered numbers that correspond to values in lists.
- Dictionaries allow you to associate one value with another in a list.
- Key-value pairs are two values in a dictionary that correspond to one another.
- Abstractions are simplifications of more complicated ideas.

## ▼ Lecture 3: Exceptions

### 11/08/23

- A syntax error is  a problem with the code that you have typed.
  - They are usually caused by typos.
- Runtime errors happen sometime when the program is being run.
  - One should write programs defensively to avoid errors.
- `try` and `except` are used in Python for error handling.
  - This example catches a mismatch of values, a value error:

```
# Try to execute code
try:
    x = int(input("Input: ")
# In case of a value error
except ValueError:
    # Value error exception
```

- Name errors can happen when the name of a variable is not defined.
  - `else` can be used if no exceptions are thrown:

```
# Try to execute code
try:
    x = int(input("Input: "))
# In case of a value error
except ValueError:
    # Value error exception
# Without an else, a name error could be thrown is 'x' isn't defined
else:
    print(x)
```

- If one wants to handle an exception, but not do anything with the exception, `pass` can be used to try again:

```
# Try to execute code
try:
    x = int(input("Input: "))
# In the case of a value error
except ValueError:
    # Go back to try if there is nothing to do
    pass
```

### Definitions

- Exceptions are problems in a program.
- Syntax determines how a programming language is written.
- Literals are constant values assigned to variables.

## ▼ Short 1: Debugging

### 11/09/23

- `print()` statements can be a “quick and dirty” way to debug code.

```
# See output of i in the console
for i in range(n):
    print(i)
    # Execute other code
```

- Breakpoints can be used to see what is happening on a certain line, and the value or variables at that point in time.

### 11/13/23

- The debugger in VS Code can be run by clicking “Run and Debug”.
    - This will run the program as usual, but pause at a breakpoint.
- “Step over” (line pointing over a dot), allows one to step over a function and continue to the next line.
- “Step into” (line pointing down at a dot), allows one to step inside a function.
- While debugging, the existence and value of variables can be tracked under the *Variables* dropdown in VS Code.
    - Variables that are “local” to a function will be under the *Locals* dropdown.
    - Variables that are “global” in a program will be under the *Globals* dropdown.

### Definitions

- Debugging refers to getting rid of bugs in a program.
- Breakpoints are mechanism that specify what line one wants to pause or ‘break’ the execution of the code.

## ▼ Lecture 4: Libraries

## 11/14/23

- Libraries can be shared across Python using modules.
  - Modules encourage reusability of code.
- Python includes some base modules.
  - Documentation for these modules can be found in the official Python documentation:

    http://docs.python.org/3/library

- `import` can be used to import the contents of a module:

```
# Import the 'random' library
import random
```

- `from` can be used to import a specific function or group of functions from a modules:
  - This is especially useful, for example, when the library being used is very large and importing the entire library is not necessary.
  - Scoping to only the function makes the code shorter.

```
# Import the function 'choice' from the 'random' library
from random import choice
```

- Command-line arguments allow inputs or “arguments” to be executed in the command line:

```
# Program code
# Import sys.argv
from sys import argv
# Include the second argument in the command line prompt
print("Hello, my name is", argv[1])

# Command line prompt
$ python name.py Name
```

- Double quotes can be used to combine multiple arguments into one:

```
# Treat the first and last name as a single argument
$ python name.py "First Last"
```

- `sys.exit()` can be used to exit the program prematurely, with an optional message:

```
sys.exit("Print exit statement")
```

- A list can be slices in python with square brackets and a colon:

```
# Slice the list starting at 'n' index
list_slice = list[n:]
# Slice the list starting at index 'x' and ending at index 'y'
list_slice = list[x:y]
# Slice the list ending at the 'n'th index from the end
list_slice = list[:-n]
```

- Packages are third party libraries contained in folders.
    - Packages can be found at:

PyPI · The Python Package Index

The Python Package Index (PyPI) is a repository of software for the Python programming language.

https://pypi.org/

- Packages can be imported using *pip install packages* (pip) in the command line:
    - This is  a recursive acronym!

```
# Install the package 'cowsay'
$ pip install cowsay
```

- Application programming interfaces (APIs) can be interacted with through code.
    - These APIs are usually hosted online on servers.
    - For example, the *requests* library allows one to make internet requests with code.
- JSON is "language agnostic", meaning it doesn't matter what programming language one is using to interpret the data.
    - The *JSON* library allows JSON strings to be formatted is a more readable way in Python.
- One has the ability to make one's own libraries.
- `__name__` is a special variable that is set to be `__main__` only when that file is run from the command line:
    - In other words, it will only be set to main when that file is the source file.
    - When the module is imported, it will be set to the name of the module instead.

```
# Aviod calling main when importing the module to another program
if __name__ == "__main__":
# Test functions in the module
    main()
```

## Definitions

- Libraries are files of code that can be used in another program.
- Modules are libraries that have one or more functions or features built into it.

- Command-line arguments allow arguments from the command line to be used in a program.
- Slicing refers to getting a subset of a list.
- Packages are third party libraries. Specifically, they are modules that have been implemented in whole files instead of just one folder.
- Application programming interfaces (APIs) are third party services that one can write code to interact with.
- JavaScript Object Notation (JSON) is a “language agnostic” text-based format for exchanging data between computers.

## ▼ Short 2: Style

### 11/17/23

- Python has a codified style called Python Enhancement Proposal 8 (PEP 8).
    - It tries to standardize what Python code looks like.

https://peps.python.org/pep-0008/

- PEP specifies:
    - “Readability counts”.
    - Indentation: 4 spaces.
    - Maximum line length: 79 characters.
    - Blanks lines: 2 blank lines should surround each function.
    - Imports: should be on separate lines.
- `pylint` is a linter that can be installed with pip:

```
# Install Pylint
$ pip install pylint
```

- `pycodestyle` is a formatter that can format ones code according to PEP 8.
- `black` is an increasingly more popular formatter:

```
# Install Black
$ pip install black

# Format a program
$ black program.py
```

### Definitions

- Style describes the way code is formatted.
- Linters find mistakes or inconsistencies in a program.

- Formatter change a program's style to a specified style.
- Opinionated code means that the language specifies a 'correct' way to write programs.

## ▼ Lecture 5: Unit Tests

### 11/17/23

- Unit tests are programs that test code from other programs.
  - For example, `test_program.py` would run a unit test on `program.py`.
  - For example, the function `test_func()` would run a unit test on `func()`.
- Corner cases can make function appear to be working correctly when they may actually not be.
  - This is why it is important to run multiple cases on a program or function.

### 11/18/23

- `assert` is a keyword in Python that allows one to 'assert' if something is true.
  - If `true`, nothing happens.
  - If `false`, an error message will appear.

  ```
  # Assert example
  assert 2 * 2 == 4
  ```

- An `AssertionError` will be thrown if assert returns false.
- Instead of writing a whole new large program to test one's code, a library like `pytest` can be used to automate this process.
  - The tests still need to be written, but `pytest` deals with the error handling.
- Breaking your code into smaller functions allows for easier unit testing.
- `pytest` can be used to catch other errors besides `AssertionError`:

```
# Import pytest
import pytest

# Catch a type error
with pytest.raises(TypeError):
    int("string")
```

- Functions are easier to test if they have a return value rather than a side effect.
- Tests should be simple and small so the tests themselves don't fail.
- `pytest` and other testing libraries support putting multiple tests in a subfolder.
  - Putting a file called `__init__.py` in that subfolder tells Python to treat that folder as a package.
    - A package is a collection of modules that are organized in a folder.

```
program.py
# Test folder
test
+-- __init__.py
+-- test_func1.py
+-- test_func2.py

# Run pytest on all the tests in the folder
$ pytest test
```

- Unit tests are important to make sure that if code is changed, the overall function of the code is not broken.

## ▼ Lecture 6: File I/O

### 11/18/23

- File I/O is about writing code that can read information from or write information to files persistently.
    - Persistence refers to the continuance of data even when a program ceases to continue running.
- `open` opens a file so that information can be written to/read from it.

```
# Open a file to write to
# Assign that file to a variable
file = open("file.txt", "w")

# Write data to the file
file.write(data)

# Close the file
file.close()
```

- `"w"` overrides the contents of a file. To append, use `"a"`:

```
# Open a file to append to
file = open("file.txt", "w")

# Append data to the file
file.write(data)

# Close the file
file.close()
```

- `with` allows one to specify that within a scope, the file should be automatically opened and closed:
    - This removes the need for the file to be closed manually.

```
# Append data to a file
with open("file.txt", "a") as file:
    file.write(data)
```

- One can read all of the lines from a file and return it as a list like so:
  - `"r"` is the implicit value for that argument, so it does not necessarily need to be specified.

```
# Store lines from a file in a list
with open("file.txt", "r") as file:
    lines = file.readlines()

# Print out the lines
for line in lines:
    print(line)
```

- Files can be iterated through line-by-line using loops:

```
# Iterate through each line in a file
with open("file.txt", "r") as file:
    for line in file:
        print(line)
```

- Instead of using a `.txt` (text) file, one can use a `.csv` (comma separated value) file to store multiple related pieces of information in the same file.
  - This format is commonly used in Microsoft Excel and other spreadsheet applications.

```
# Iterate through each line in a file
with open("file.csv", "r") as file:
    for line in file:
        # Assign the line into a list, with value seperation denoted by commas
        row = line.split(",")

# Access each value in the row
var_1 = row[0]
var_2 = row[1]
var_n = row[n-1]
```

- Variables from a list can be unpacked:

```
# Instead of assigning the line to a list
row = line.split(",")
# One can unpack the values in the line
var_1, var_2, var_n = line.split(",")
```

- Functions can be passed into other functions as arguments.

    - This can be used, for example, to tell the `sorted()` function how to sort the contents of a list.
        - The function passed into `key=` is automatically called by `sorted()` on each of the dictionaries in the list.
        - It uses the return value to of the passed in function to decide what strings to use to sort the list.

```
# This function specifies to the sorted() function
# which part of each dictionary to use to sort the list
def get_name(student):
    return student["name"]

# Sort the students in the list based on their names
for student in sorted(students, key=get_name):
    print(student["name"])
```

- A lambda function is an inline function that has no name.
    - These are usually used when a function is only needed in one place.

```
# Example of a lambda function that is equivalent to the get_name() function above
for student in sorted(students, key=lambda student : student["name"]):
    print(student["name"])

# Anatomy of a lambda function
lambda parameters : return_value
```

- The `csv` module can be used to streamline the task of reading from and writing to `.csv` files:

```
# Import csv module
import csv

# Open the .csv file
with open("file.csv", "r") as file:
    # Create a csv reader for the file
    reader = csv.reader(file)
    for row in reader:
        # Iterate through each row in the file using the reader
```

- `csv.DictReader` can be used in place of `csv.reader` to create a dictionary of columns rather than a list of columns
    - The names of the columns can be explicitly stated in the first line of the `.csv` file.
    - This makes the `.csv` data more resistant to the columns being switched around.
    - A dictionary reader can change over time but the existing code is not going to break.
- `.csv` data containing commas can be put in quotes to distinguish that the comma is not a separator

```
name, greeting
# Greeting is in quotes to keep it as one data point
Nate,"Hello, how are you"
```

- `csv.DictWriter` can be used to write to a file:

```
# Write to a .csv file
with open("file.csv", "a") as file:
    # Assign a dictionary writer to the file
    writer = csv.DictWriter(file, fieldnames=["name", "greeting"])
    # Write to a row
    writer.writerow({"name": name, "greeting", greeting})
```

- A binary file or 'bin' is a file that is just 0s and 1s.
- Image files can be written to and read from.
    - One example of an image input/output library is Pillow, or `PIL`.

https://pillow.readthedocs.io/en/stable/

- Example of how to make a simple 2 frame GIF:

```
# Create a simple 2 frame GIF

import sys
# Import 'Image' from PIL
from PIL import Image


images = []


# For each argument in the command line
# Excluding the first argument
for arg in sys.argv[1:]:
    # Open each image file
    image = Image.open(arg)
    # Append each image to images list
    images.append(image)


# Create the GIF
images[0].save(
    # Name of file
    "costumes.gif",
    # Save images
    save_all=True,
    # Images to append to starting image
    append_images=[images[1]],
    # 200ms per image
    duration=200,
    # Loop infinitely
    loop=0
)


# Create GIF
$ python costumes.py costume1.gif costume2.gif
# Run GIF
$ code costumes.gif
```

## ▼ Lecture 7: Regular Expressions

### 11/19/23

- Regular expressions or `regexes` are patterns that can be used to compare, validate, or clean up data.
- In Python, there is a library for regular expressions called `re`.

https://docs.python.org/3/library/re.html

- `re` comes with a list of `regex` pattern symbols:

```
# Any character except a newline
.
# 0 or more repetitions
*
# 1 or more repetitions
+ or .*
# 0 or 1 repetition
?
# m repetitions
{m}
# m-n repetitions
{m,n}
# Match on a dot (because '.' usually means what is listed above)
\.
# Match on a double quote (because '"' usually encapsulates a string)
\"
# Matches the start of the string
^
# Matches the end of the sring or just before
# the newline at the end of the string
$
# Set of characters rather than any character
[]
```

```
# Complementing the set (everything except these charcters)
[^]
# Set of characters from A through B
[A-B]
# A or B
A|B
# Group charcters together (order of operations)
(...)
```

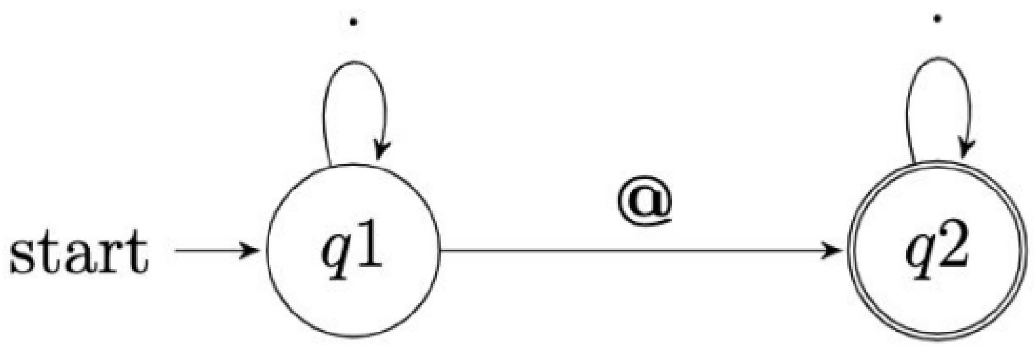
- Some `regex` patterns are so common, there are built in syntax patterns that can be used in their place:

```
# Instead of matching [a-zA-Z0-9_], matching a word charcter
\w
# Not a word charcter [^a-zA-Z0-9_]
\W
# Decimal digit [0-9]
\d
# Not a decimal digit [^0-9]
\D
# Whitespace charcter
\s
# Not a whitespace character
\S
# Word boundry
\b
# Not a word boundry
\B
```

- For example, these patterns can be used to validate strings:
  - the `r` tells Python to interpret the string as a raw string, meaning it should not interpret any backslashes in the usual way.
  - Flags are configuration options that allows one to configure a function differently.

```
# Check if a .edu email is valid:
if re.search(r"^\w+@(\w+\.)?edu$", email, re.IGNORECASE):
    # Execute code if .edu email is valid
```

- Regular expressions use finite-state machines to detect patterns:
  - The computer goes through each character sequentially from left to right.
  - If the computer finds itself in the double circle at the end of the string, it confirms the string’s validity.



- Regular expression searches can capture data are return it as a variable:

```
# Everything in the parenthesis are returned from re.search
(...)
# Non-capturing group
(?:...)
```

```
# Separate first and last name
matches = re.search(r"^(.+), *(.+)$", name)
# If the string matches the regex pattern,
# reformat the name string accordingly
if matches:
    # Split the matches into seperate variables
    last, first = matches.groups()
```

- The `:=` operator, or "walrus operator", can be used if one wants to assign a value, but also ask a conditional on the same line:

```
# Separate first and last name
if matches := re.search(r"^(.+), *(.+)$", name):
    # Split the matches into seperate variables
    last, first = matches.groups()
```

- `re.sub` substitutes characters in a string based on a `regex` pattern.
- Regular expressions should be, much like any program, written incrementally.
  - This makes it easier to catch errors and improve patterns over time.

## ▼ Lecture 8: Object-Oriented Programming

## 11/19/23

- Object-oriented programming is a paradigm of programming that organizes code into classes.
    - A paradigm is a way to classify a program based on its features.
- Up until now, this course has been using functional programming, a way of structuring programs into functions.
- In Python, a function can return multiple values in what is called a tuple:
    - A tuple is a collection of values that is immutable
        - Immutable is a data structure that cannot be modified.

```
# Return the values of both x and y in a tuple
def func(n):
    x = n * 2
    y = n + 1
    return x, y
```

- Only one value is really being returned, that is, the tuple datatype is being returned.
- Parenthesis can be put around the tuple to make it more explicit that only one value is being returned:

```
# Show visually that only one value (a tuple) is being returned
return (x, y, z)
```

- If a returned tuple is assigned to a single variable, the values of the tuple can be accessed like a list:

```
# Assign name and house to a student variable
student = get_student()
# Print the student's information
print(f"{student[0]} from {student[1]}")


def get_student():
    return (name, house)
```

- In Python, one can state that they want a list to be returned instead of a tuple:

```
# Return the values of both x and y in a list
def func(n):
    x = n * 2
    y = n + 1
    return [x, y]
```

- Classes allow one to make their own custom datatypes.

https://docs.python.org/3/tutorial/classes.html

- This is an example of a simple `Student` class:
    - Classes have attributes (technically, instance variables), properties that allow one to specify values inside the class.
        - Class attributes can be accessed using the `.` operator.
        - Class names, by convention, should be capitalized.
    - An object is the instantiation or instance of a class datatype.
        - An object instance is to arguments as classes are to parameters. The former exists only during runtime.
    - Classes can be made mutable or immutable.
    - Methods are functions that can be specified inside a class.
        - Or more generally, when a function is inside a class, it is called a method.
    - The `__init__()`, or "dunder init" method is a built in instance method, used to initialize the contents of an object from a class.
        - Dunder refers to the "double underscore" `__`.
        - It is the class constructor, which instantiates an object.
            - Python technically calls this constructor the "initialization method".
    - `self` is a reference to the current object itself. This allows a place to store instance variables.
        - It essentially "installs" the passed in arguments to the instance variables in the current object.

```
# Simple student class
class Student:
    # Constructor
    def __init__(self, name, house):
        # Assign instance variables to current object
      self.name = name
        self.house = house


# Create an instance of the student class
student = Student("My Name", "My House")
# Print student information
print(f"{student.name} from {student.house}")
```

- In Python, `...` is a placeholder that can take the place of an empty line

```
# A function that does nothing
def func():
    ...
```

- The `raise` keyword allows one to "raise" their own exceptions:
    - `raise` gives the programmer more control over when they want errors to be raised during runtime.

```
# Raise a ValueError
raise ValueError

# Raise a ValueError and print to the terminal
raise ValueError("Value Error!")
```

- `__str__` is a built in, customizable method in Python which can do something when the object is asked to be converted to a `str`:

```
# Simple student class
class Student:
    # Constructor
    ...


    # Dunder string
    def __str__(self):
        return "a student"
```

- One can defined methods inside a class:
    - Recall that `self` needs to be passed, if needed, to reference the object.

```
# Simple student class
class Student:
    # Constructor
    ...


    # Dunder string
    def my_method(self):
        ...
```

- Getters are functions from a class that “get” an attribute
- Setters are functions from a class that “set” an attribute.
    - These getters and setters override the default functionality, if they are named according to the correct attribute.
    - Decorators are functions in Python that modify the behavior of other functions.
        - Two of these are `@property` for getter methods and `@attribute.setter` for setter methods.
        - The property is essentially a custom attribute defined by the corresponding instance variable’s setter method.

```
# Getter method
@property
def house(self):
    return self._house
```

```
# Setter method
@house.setter
def house(self, house):
    self._house = house


...


# Default functionality is overriden and uses the methods above
student.house = "My House"
```

- By convention, `_` are put in front of instance variable names in getter and setter methods to distinguish them from the method name:
    - Even attribute assignments in `__init__` use the setters. The actual attributes are the ones that start with `_`.

```
# Distinguish instance variable "_house" from  method name "house"
@property
def name(self):
    return self._name
```

- In Python, there are no permissions built into classes.
    - It depends on the honors system. If an attribute starts with `_`, it is expected that the programming should not interact with it directly.
- Class methods are methods that have the same functionality no matter what the object's instance variables are.
    - `@classmethod` is a decorator that specifies a class method that does not have access to `self`.
    - The class not not need to be instantiated as an object, it really only needs to be a container for data.
        - A class method can be called without instantiating an object first.
    - This means that the class doesn't need an `__init__` method.
    - `cls` is a reference to the class, where `self` is a reference to the current object.
    - Instance variables and methods operate on specific objects, where class variables and methods operate on the entire class.

```
# Define class
class MyClass:

    # Class method
    @classmethod
    def my_method(cls):
        return x
```

```
# Use class without instantiating it
MyClass.my_method
```

- Inheritance allows classes to “inherit”, or borrow, variables or methods from another class.
  - A parent class is known as a superclass and the child class is known as the subclass.
  - `super()` in the initializer method is a reference to the superclass.

```
# Parent class
class Pet:
    def __init__(self, name):
        self.name = name


# Child class
class Dog(Pet):
    def __init__(self, name, legs)
        # Reference to superclass
        super().__init__(name)
        self.legs = legs


# Child class
class Bird(Pet):
    def __init__(self, name, wings)
        # Reference to superclass
        super().__init__(name)
        self.wings = wings


# Create a pet object
pet = Pet("Perry")
# Create a dog object
dog = Dog("Fluffy", 5)
# Create a bird object
bird = Bird("Tweety", 4)
```

- Operator overloading allows one to implement custom functionality for operators using built in overloading methods.
  - For example, `+` has a built in overload for concatenation.
  - Unfortunately, there is no way to define custom overloading methods in Python specifically.
  - For `__add__` and other overloads, the first operand is `self` and the second operand is `other` by convention.

```
# Coins class
class Coins:
    def __init__(self, pennies, nickles, dimes, quarters)
        # Assign coin amounts
        ...

    # Overload the "+" operator
    def __add__(self, other)
        pennies = self.pennies + other.pennies
        ...
        quarters = self.quarters + other.quarters
        return Coins(pennies, nicles, dimes, quarters)


# Create two wallet objects
wallet_1 = Coins(5, 2, 3, 7)
wallet_2 = Coins(3, 5, 8, 9)
# Add the coins in both wallets to create a third wallet object
wallet_combines = wallet_1 + wallet_2
```

- The list of Python operator overloads can be found here:

https://docs.python.org/3/reference/datamodel.html#special-method-names

## ▼ Lecture 9: Et Cetera

### 11/20/23

- `set` is a datatype like a list that eliminates duplicate values.
- A global variable is a variable that is at the top level of the program.
- A local variable is a variable that can only be accessed within a certain scope.

### 11/21/23

- Global variables can be only modified by functions in a program if the keyword `global` is used.

```
# Global variable
var


def main():
    func1()
    func2()


# Balance cannot be modified
def func1():
    var += 1
```

```
# Var can be modified
def func2():
    global var
    var += 1
```

- Many languages allow one to set variables with a `constant` keyword, so that variables that cannot be changed.
    - Python does not have this keyword, so constants are `CAPITALIZED` by convention.
        - They can be still modified, so this style runs on the honors system.

```
# One can modify this variables
var


# One should not modify this variable
VAR
```

- For methods inside a class, constants should use the `ClassName.CONST` rather than `self.CONST`.
- Type hints check if variables in a program are using the right datatypes.
    - One library that can check types is `mypy`.

https://docs.python.org/3/library/typing.html

```
# Catch type error
def square(n: int):
    return n * n

# Catch type error
x: int = input("What's x?")
x_sqaured = square(n)
```

```
# Run mypy
$ mypy program.py
```

- `mypy` can also check return types by hinting `->`:

```
# Catch return value
def square(n: int) -> int:
    return n * n
```

- `docstrings` or document strings are standardized ways in Python that one should document strings.
    - Docstrings are surrounded by triple quotes `"""`:

    - Docstrings allow for better and standardized documentation of code.

```
# Function with a docstring
def func_simple():
    """Docstring explaining what func_simple() does"""
    ...


# More complex function with a docstring
def func_complex(n: int) -> str:
    """
    Description of func_complex().

    :param n: Parameter n explaination
    :type n: int
    :raise TypeError: func_complex could raise a TypeError if something happens
    :return: What func_complex returns
    :rtype: str
    """
    ...
```

- `argparse` is a default Python library that handles command line argument parsing automatically.

https://docs.python.org/3/library/argparse.html

```
# Import argsparse
import argparse


# Create parser
parser = argparse.ArgumentParser()
# Take "-n" command-line argument
parser.add_argument("-n")
# Get command-line arguments
args = parser.parse_args()

# Get "n" from command line argument after "-n"
for _ in range(int(args.n)):
  print("hello")
```

- `-h` or `--help` can be used after a program with `argparse` to show the usage of a program.

```
# Run help
$ python program.py -h
```

```
# Command line output
usage: program.py [-h] [-n N]

options:
  -h, --help  show this help message and exit
  -n N
```

- A description and help parameter can be added to `argparse` functions to give more info when `-h` or `--help` is run.

```
# Create parser
parser = argparse.ArgumentParser(description="Say hello")
# Take "-n" command-line argument
parser.add_argument("-n", help="number of times to say hello")
```

- A default and datatype can be specified for an argument.

```
# Take "-n" command-line argument
# This code is indented for easier reading. In an actual program,
# it should all be on one line
parser.add_argument(
    "-n",
    default=1,
    help="number of times to say hello",
    type=int
)
```

- Unpacking is the process of extracting multiple variables from a function:

```
# Unpack multiple variables from a return value
first, last = input("What's your name? ").split(" ")
```

- A `*` can be used at the beginning of a list to unpack it:

```
# Unpack coins into a function that takes 4 arguments
def add_coins(pennies, nickles, dimes, quarters):
    ...


coins = [10, 25, 13, 7]
add_coins(*coins)
```

- A `**` can be used at the beginning of a list to unpack it:

```
# Unpack coins into a function that takes 4 arguments
def add_coins(pennies, nickles, dimes, quarters):
    ...
```

```
coins = {"pennies": 10, "nickles": 25, "dimes": 13, "quarters": 7]
add_coins(**coins)
```

- `*` can be used in a parameter to define a function to take a variable number of positional arguments.
  - These arguments are converted to a single tuple.
- `**` can be used in a parameter to define a function to take a variable number of positional keyword arguments.
  - These keyword arguments are converted to a single dictionary.
- Functions, by convention, use the names `arg` and `kwargs` by convention.

```
# A function that takes any number of arguments and keyword arguments
def func(*arg, **kwargs):
    ...


# Pass in four arguments and two keyword arguments
func(100, 50, 25, 5, first="First", last="last")
```

- The `map()` function allows one to apply, or “map” some function to every item in a sequence like a list:

```
# Prints USE YOUR INSIDE VOICE
words = ["Use", "your", "inside", "voice"]
# Capitalize every string in the list
yell = map(str.upper, words)
print(*yell)
```

- list comprehensions are an alternative, more “Pythonic” alternative to mapping:

```
# Prints USE YOUR INSIDE VOICE
words = ["Use", "your", "inside", "voice"]
# Capitalize every string in the list with a list comprehension
yell = [word.upper() for word in words]
print(*yell)
```

- List comprehensions can have conditionals:

```
nums = [1, 4, 5, 7]
# Multiply every number in a list by 2,
# and only filter through numbers less than 10
double_nums = [num * 2 for num in nums if num < 10]
```

- `filter()` can be used in a similar sense to `map()` and filter a sequence like a list based on a function that returns a `bool`:

```
# Returns a True or False value
def is_less_than_10(n):
    return n < 10


nums = [1, 4, 9, 11]
# and only filter through numbers less than 10
nums_under_10 = filter(is_less_than_10, nums)
```

- Dictionary comprehensions can be used to map and filter dictionaries:

```
# Assign the Harry Potter characters to Gryffindor,
# with their names and houses as key-value pairs
students = ["Hermione", "Harry", "Ron"]
gryffindors = {student: "Gryffindor" for student in students}
```

- `enumerate` can iterate over a sequence and get the value and index of each item:

```
# Print student names in an ordered list
students = ["Hermione", "Harry", "Ron"]

for i, student in enumerate(students):
  print(f"{i + 1}. {student}")
```

- Generators functions can be used to return sections of data at a time rather than all of the data from a function at once using the keyword `yield`.

https://docs.python.org/3/howto/functional.html#generators

```
# Return one number at a time
def square(n):
    for _ in range(n):
        yield n * n

# Print the first 1,000,000 non-negative integers
print(square(1000000))
```

- `return` returns a value, but `yield` returns an iterator.
    - An iterator can be stepped over in a loop one element at a time. On each iteration, `yield` only returns the one value that is appropriate for the current index of the iterator.