# Intro To Python

Day 1

Prepared By: Jonathan Schlosser

# What is a Program?

A program is a set of instructions that the computer follows to perform a task or to solve a problem.

This definition is quite broad. As such, any number of things can be a program.

Programs range from just a couple lines of code, to thousands of lines of code in robust repositories.

# Let's Try An Exercise!

You are tasked with writing out a "program" that would make a Peanut Butter and Jelly Sandwich.

Think about the step-by-step instructions would one need if they had never encountered a peanut butter and jelly sandwich before.

How would you communicate this process? What would the steps be? What assumptions are needed for those steps? Etc.?

# So, how does this relate to a computer program?

We need to understand that:

- Computers are pretty dumb.
- They will do exactly what you tell them to do.
- They take your instructions literally.
- They will do **exactly** what you tell them to in the **order** you tell them to do it.
- They will do things really fast, and will appear smart, but they will only be doing what you said.
- And, computers do not remember things, unless you tell them how to remember.

# Some Background on Computers:

- A computer understands 1's and 0's, or sequences of 1's and 0's.
- A central processing unit (CPU):
  - Reads instructions from memory.
  - Decodes instructions to determine which operations to perform.
  - And, performs the operations.
- What is an operation?
  - Operations are any action a computer may undertake.
  - This can include reading data, adding, subtracting, multiplying, dividing, etc.

# Often, we don’t work in 0’s and 1’s

This machine language is hard to use.

- It is practically unreadable.
- It is hard to maintain and debug.
- There are no mathematical functions available.
  - We would need to create our own code for these routines every time we wrote a program.
- Memory locations are manipulated directly.
  - Which would require a programmer to keep track of every memory location on the computer!
  - And would be directly affected by the available computer memory!

With this, we use an english like language (and a converter) to communicate with the computer.

# Languages

High Level Languages include:

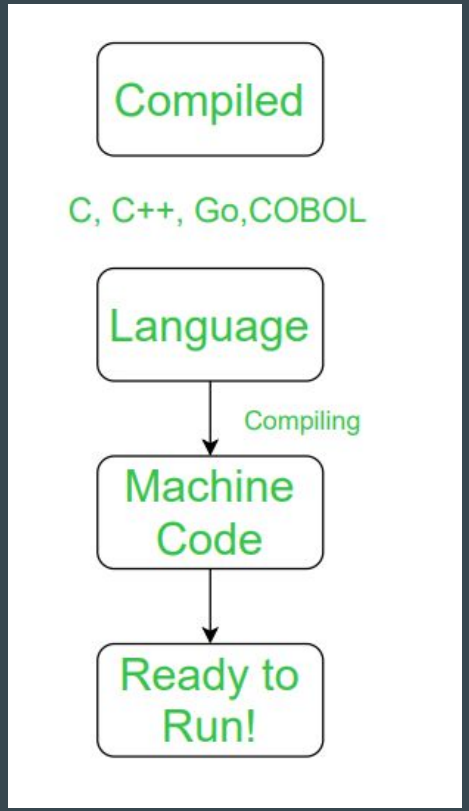
- Compiled: Java, C#, C++, Fortran, Objective-C, Rust, Swift
- Interpreted: Python, PHP, Javascript, Perl, Ruby

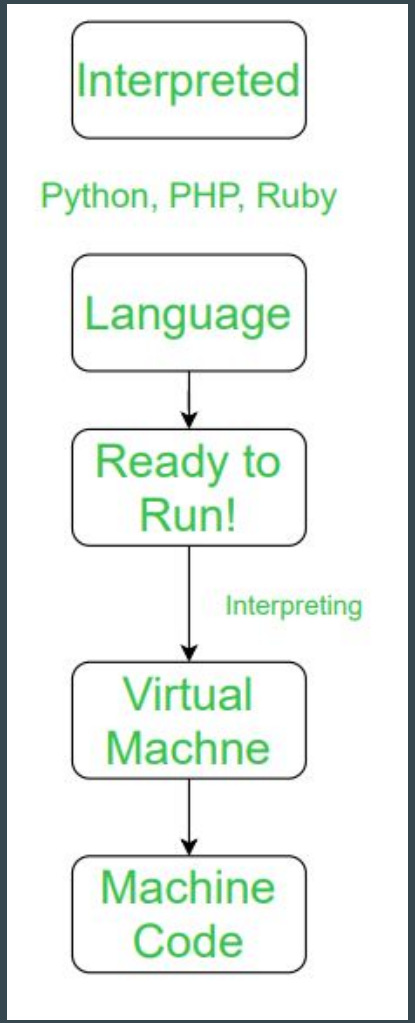
Compiled languages employ a compiler to translate and package the code into machine language.

- In this language, once the program is compiled it is expressed in the instructions of the target machine. This makes it much faster, but unreadable by humans.

Interpreted languages employ an interpreter to execute instructions directly and freely, without previously compiling the program into machine-language instructions.

- The code is interpreted at run-time. This makes it a bit slower, but allows for more flexibility in terms of modifying code and debugging.


Compiled
C, C++, Go,COBOL
Language
Compiling
Machine Code
Ready to Run!


Interpreted
Python, PHP, Ruby
Language
Ready to Run!
Interpreting
Virtual Machne
Machine Code

# Python

History

- Guido van Rossum (Benevolent Dictator for Life)
- Started implementation in 1989


Philosophy

- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated.
- Readability counts.
- Python should be fun to use.

# Python

Features:

- Very clear and readable syntax
- Strong introspection capabilities
- Intuitive Object Oriented
- Modular
- Exception-based error handling
- Very high level dynamic data types
- Extensive standard library

Main Python Site: http://www.python.org/

# Foundations

# Ways to work with Python

Interactive Shell:

- This is where you would directly enter commands and have an output returned.
- It is immediate and only really used for small bits of code.

In a Script:

- This is where we would create a Python script with a .py extension.
- Essentially here we are just saving Python statements in a file.

In an Integrated Development Environment (IDE):

- An IDE is an environment that provides tools to write, execute, and test a program.
- We will be using Jupyter Notebooks and Spyder for this.

# Python has 29 keywords...

and, def, exec, if, not, return, assert, del, finally, import, or, try, break, elif, for, in, pass, while, class, else, from, is, print, yield, continue, except, global, lambda, raise

These are the built-in keywords for Python.

They have special meanings and represent the foundational Python language.

But, Python works with many **libraries** that drastically expand its functionality, including common packages like NumPy and Pandas.

# Hello World Example

print(‘Hello World’)

- Print is a keyword.
- ‘Hello World’ is an argument passed to the print function.
- And the output of Hello World is the value.

Values have types in python.

- Here, Hello World is a string.
- A string is a sequence of characters.
- A string can be identified in Python if it is wrapped in quotes.
- Other types of values include integers, float, lists, dictionaries, etc.

# Variables

- A variable is a name that refers to a value.
  - It is just a name that represents a value stored in the computer memory.
- We create variables with an **assignment statement.**
  - An assignment statement is used to create a variable and to declare the information it is referencing.
  - In an assignment statement, the variable receiving the value must be on the left side.
  - The general format is: variable = expression
    - The equal sign (=) in Python is the assignment operator.
    - Example:
      - age = 29
      - a_string = ‘Hello World’
      - aFloatNumber = 1.437
  - You can only use a variable if a value is assigned to it.

# Rules For Variables

- There are a number of rules we must follow when creating variables.
  - Variable names cannot be a Python keyword.
  - Variable names cannot contain spaces.
  - The first character must be a letter or an underscore.
  - After the first character, names can include letters, digits, or underscores.
  - Variable names are case sensitive.
  - Variable names should reflect their use; they should be informative.
- Variables can be:
  - Printed
  - Reassigned
  - Called
  - And Manipulated

# Statements

A statement is an instruction that the Python interpreter can execute.

There are many types of statements in Python.

Python reads statements line by line.

Examples include:

- The print statement.
- The assignment statement.

# Comments

Comments are brief notes that are included in a program to help explain or guide the understanding of the code.

Comments are ignored by the Python interpreter and are intended for the people reading the code.

They begin with a # character.

Sometimes, comments can be included at the end of a line to explain the purpose of that line; these are end-line comments.

# Expressions

Expressions are a combination of values, variables, and operators.

Operators are special characters that represent computations.

- Addition (+)
- Subtraction (-)
- Division (/)
- Multiplication (*)
- Exponentiation (**)
  - Raises a number to a power ($x^y$)
- Remainder or Modulus (%)
  - Performs a division operation and returns the remainder.
  - Typically used to convert times and distances, and to detect odd or even numbers.

Expressions follow the order of operations (PEMDAS).

# The Input Keyword

- Most programs need to read input from a user, and because of this, Python has a built-in input() function that reads input from the keyboard and returns the data as a string.
- Variable = input(prompt)
  - The prompt is typically a string instructing the user to enter a value.
- The prompt will be displayed exactly as it is entered.
- The input function always returns a string
- If we need a different data type, we need to convert it.
  - int(item) convert the item to an integer.
  - float(item) converts the item to a float value.
  - The conversion only works if the item is valid for the intended data type, otherwise it will bring back an exception.

# Exercise

Write a program that:

- Takes two numbers from the user.
- Prints out the following for the two numbers:
  - Sum
  - Difference
  - Product
  - Quotient

# Functions

# What is a function?

A function is a named sequence of statements that perform a computation.

Functions can be executed in order to perform an overall program task; they are usually one task of a larger program.

A modularized program is a program where each task within the program is in its own function.

Python has built-in functions and user-defined functions.

# Why would we use functions?

Functions make programs easier to read and debug.

- It allows for the grouping and organization of statements.
- And, if changes need to be made, they only need to be made in one place, within the function.

Functions also help reduce repetitive code and makes programs smaller.

Well-designed functions are often useful for many programs, and, if deployed correctly, can be used across programs.

# Built-In Functions

| | | Built-in Functions | | |
|---|---|---|---|---|
| abs() | divmod() | input() | open() | staticmethod() |
| all() | enumerate() | int() | ord() | str() |
| any() | eval() | isinstance() | pow() | sum() |
| basestring() | execfile() | issubclass() | print() | super() |
| bin() | file() | iter() | property() | tuple() |
| bool() | filter() | len() | range() | type() |
| bytearray() | float() | list() | raw_input() | unichr() |
| callable() | format() | locals() | reduce() | unicode() |
| chr() | frozenset() | long() | reload() | vars() |
| classmethod() | getattr() | map() | repr() | xrange() |
| cmp() | globals() | max() | reversed() | zip() |
| compile() | hasattr() | memoryview() | round() | __import__() |
| complex() | hash() | min() | set() | apply() |
| delattr() | help() | next() | setattr() | buffer() |
| dict() | hex() | object() | slice() | coerce() |
| dir() | id() | oct() | sorted() | intern() |

# Built-In Functions

To use a built-in function, we call them.

print(‘Hello World’) uses the print function. The input is ‘Hello World’ and the output is Hello World, or the words themselves.

len() is another commonly built-in function that works to return the length of an object. The length of an object is the number of items within that object. An argument here, or the object passed to the function, can be a sequence (a string, bytes, tuple, list, range, etc.) or a collection (dictionary, set, array, dataframe, etc.).

len(‘Hello World’) has an input of ‘Hello World’ and an output of 11.

Try to get the length of Hello World yourself and print it to the console.

# User-Defined Functions

- Format:
  - def name(inputs):
    - Statement
    - Statement
    - Statement
- The statements are indented because Python uses this positionality to determine where a function begins and ends.
  - With this each indentation needs to follow suit for that code block; in other words the tabs or spaces need to be the same.
  - You can use tabs or spaces, but not both. If you use both, it can confuse the Python interpreter.
  - Blank lines are ignored in functions.
- After defining a function, you can call it when needed like a built-in function. When it is called, the interpreter will jump to the function and executes statements in the code block.
  - A function call must come after the function definition!

# Flow of Execution

The flow of execution refers to the order in which statements are executed.

Execution always begins at the first statement of a program.

Statements are executed one at a time, in order from top to bottom.

- Example:
  - def name():
    - statement
    - statement
    - statement
  - name ()

# The main() Function

When writing a program, it is often best to define a **main()** function.

This function gets called as the last line of a program, with all of the other elements being called within the definition of this function, and runs when a program starts.

This main function calls other functions as they are needed.

It defines the mainline logic of the program.

– Twinkle.py

# Functions, Arguments, and Parameters

- When a function is called, Python will
  - Jump to the first line of a function's definition
  - Run all the lines of code within the defined function
  - And then jump back to the point where the function was called.
- When defining a function, we can declare inputs that the function will expect.
  - These are called parameters.
- When we call the function, and pass the inputs to it, these elements are called arguments.
  - These arguments get passed to the function as variables and are used within the function definition.

# Fruitful Functions and Void Functions

Fruitful functions return a value.

- These will use a **return** keyword.
- Example:
    - def add_five (number_input):
        - return number_input + 5

Void functions do not.

- Example:
    - def happy_birthday(name):
        - print('Happy Birthday', name)

# Variables

# Local Variables

- A local variable is a variable that is assigned a value within a function.
- It belongs to the function in which it was created.
- Only statements inside that function will be able to access the variable.
- If another function tries to reference this variable, an error will occur.
- A local variable cannot be accessed by statements before it is created.
  - Remember that python reads each line one at a time, sequentially.
- Also, different functions can have local variables with the same name.
  - Because the variable is within the scope of its own function, there will be no confusion with its reference.

This relates to the concept of **Scope.** Scope refers to the part of a program in which a variable can be referenced.

# Global Variables

Global variables are created by an assignment statement that is written outside of other functions.

They can be accessed by any statement within the program, including within functions.

It is **advisable** to **NOT** use global variables!

- Global variables make debugging difficult as many locations within the code could be causing an error.
- Functions that use global variables become dependent on those variables.
- And global variables make a program harder to understand.

# Global Constants

A global constant is a global variable that references a value not often changes, or a constant.

It is okay to use global constants in a program as they are often set and unchanging values.

These are constants that would be referenced throughout the program and will not need to change based on a program or a statement.

# If/Else Statements

# Boolean Expressions and Relational Operators

- A Boolean expression is tested by an if statement to determine if it is true or false.
- A relational operator determines whether a specific relationship exists between two values.
- Strings are best compared using the == and != operator, although they can be compared using the others, it would just use ASCII values.

| Expression | Meaning |
|---|---|
| x > y | Is x greater than y? |
| x < y | Is x less than y? |
| x >= y | Is x greater than or equal to y? |
| x <= y | Is x less than or equal to y? |
| x == y | Is x equal to y? |
| x != y | Is x not equal to y? |

# Boolean Variables

Boolean variables are similar to boolean expressions but are a data type in their own right.

A Boolean variable has a value of either True or False and is represented by the bool data type.

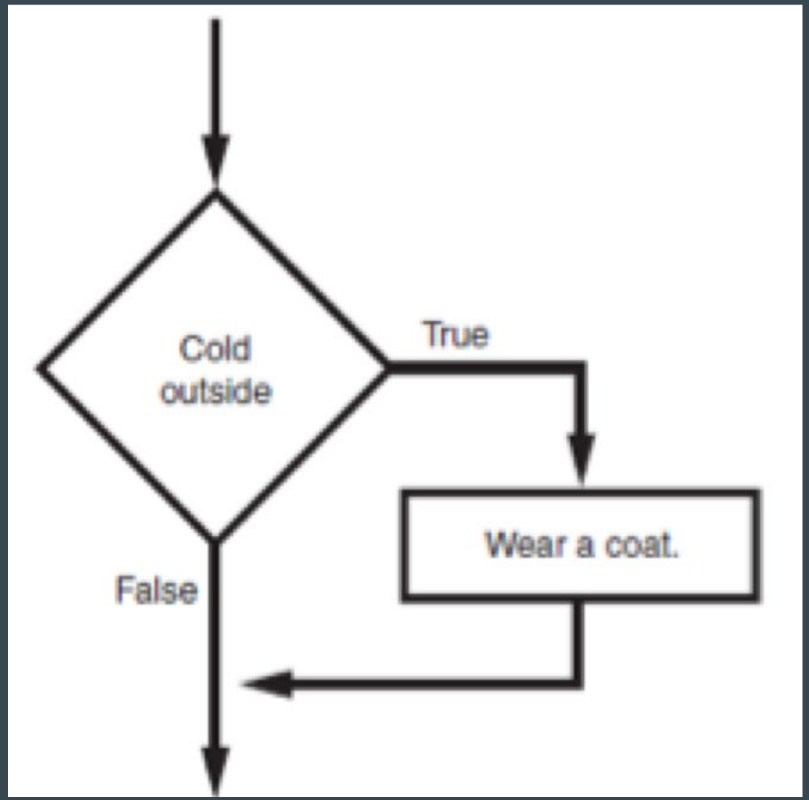
Boolean variables are often used as flags, or variables that signal when some condition exists in a program.

If a flag is False, then the condition does not exist. If it is True, then the condition does exist.

# The If Statement

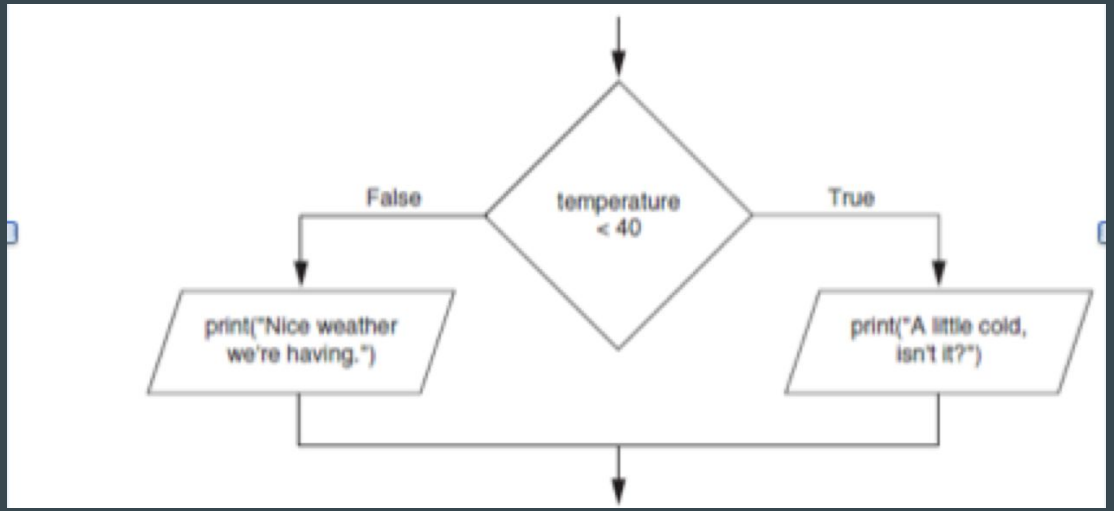
- Python Syntax:
  - if condition:
    - Statement
    - Statement
- The first line is the if clause and includes the keyword if followed by a condition.
  - The condition needs to be True or False.
  - When an if statement is executed:
    - The condition is tested.
    - If it is True, the block of statements are executed.
    - If it is False, the block statements are skipped.
- Any relational operators can be used in an if statement.
- If statements can be used within functions or independently.

# The If Statement



# The If-Else Statement

The If-Else statement is similar to the if statement, but it offers a second path as an alternative if a condition is not met.



# Nested If-Else Statements

- A decision structure can be nested inside of another decision structure.
  - In this, the first condition is checked, and if it is true, then the second condition is checked.
- When nesting statements, it is important to use proper indentation.
  - This is important for the Python interpreter.
  - It makes the code more readable for the programmer.
- Rules for Nested Statements:
  - The else clause needs to line up with its matching if clause.
  - Statements within each block need to be consistently indented.

# If-Elif-Else Statements

The if-elif-else statement is a specific decision structure that makes the logic of nested decision structures simpler and easier to write.

Syntax:

- if condition1:
  - Statements
- elif condition2
  - Statements
- Else:
  - Statements

# Logical Operators

# Logical Operators

Logical operators are operators that can be used to create complex Boolean expressions.

The “and” operator and the “or” operator are binary operators that connect two Boolean expressions into a compound Boolean expression.

- The “and” operator requires both conditions to be true, but will stop altogether if the first condition is false.
- The “or” operator requires that either condition is true, but will stop altogether if the first condition is true.

The “not” operator is a unary operator and reverses the truth of its Boolean operand.

# The in and not in operators

We can use the “in” operator to determine whether one object is contained in another object.

We can use the “not in” operator to determine whether one object is not contained in another object.

The “in” and “not in” operators are often used with strings and substrings.

“In” and “not in” can also be used to check if elements are in a list or a tuple.

# While Loops

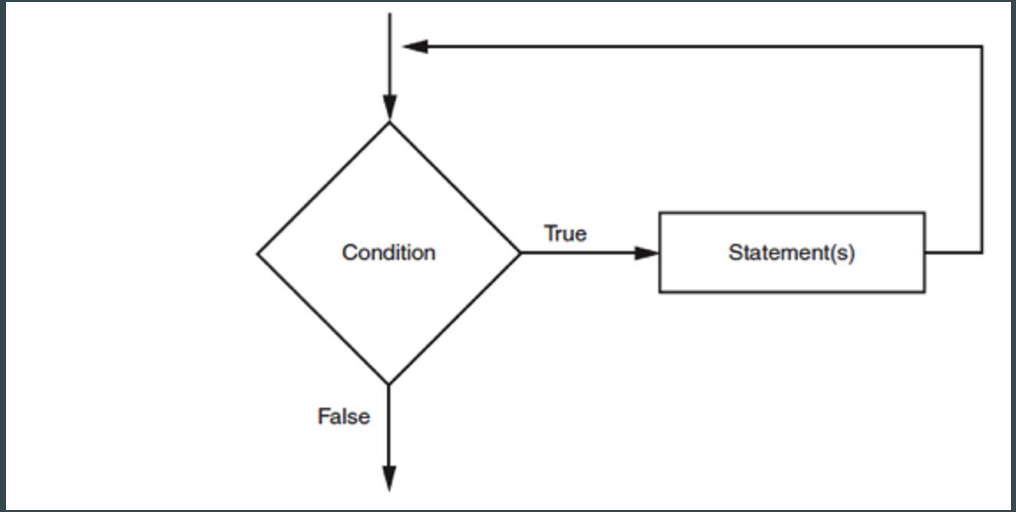
# What is a while loop?

Typical Format:

- while [test]:
  - statement
  - statement
  - statement

The *test* here is some condition that evaluates to True or False.

The indented statements are called the body of the loop.



# What is a while loop?

In order for a while loop to stop executing, something needs to happen within the loop that would ultimately make the condition false.

Loops go through iterations. An iteration is one execution of the body of a loop.

While loops must contain a way to terminate. Something inside the while loop must eventually make the condition false.

If not, you will run into an infinite loop!

An infinite loop is a loop that does not have a way of stopping. It repeats until a program is interrupted.

An infinite loop can crash a computer if it is not interrupted. So be careful here.

# Libraries and The Import Statement

# Libraries

Similar to built-in functions, like print and input, there are a number of libraries that contain pre-written functions.

These libraries are composed of modules that store functions. Modules help organize the functions within a library and help load them into our interpreter.

To call a function stored within a module, we need to write import statements.

- Import statements are written at the top of a program. They are the first statements run within a program.
- Format: import *module_name*

# Dot Notation

Dot notation is used to call a function belonging to a module.

We can use this to access specific functions within libraries.

Format: module_name.function_name()

# Random Numbers

# Generating Random Numbers

Random numbers are crucial in a number of programming tasks.

The random module is a library of functions that we can import in order to work with random numbers.

The randint function generates a random number in the range provided by the arguments.

# Lists

# Sequences

A sequence is an object that contains multiple data items.

The items are stored in sequence, one after the other.

Python has different data types that are considered sequences.

These include lists and tuples. Lists are mutable (the items in the sequence can be changed) and tuples are immutable (the items in the sequence cannot be changed).

# Lists

Lists are objects that contain multiple data items.

Each item in the list is an element.

The format of a list is: list = [item1, item2, item3, etc.]

The brackets are a special type of character in python that indicates a list data type.

Lists can hold different types of items.

And the list() function can convert certain types of objects to lists.

We can use the append() function to add items to a list.

# Indexing

Lists have an index in Python.

An index is a special number that specifies the position of an element in a list.

The index of the first element in a list in Python is 0.

The second element is 1, and so on.

Negative indexes identify positions relative to the end of the list.

The index -1 identifies the last element, -2 the second to last, and so on.

# Index Errors

An IndexError exception will occur if we try to use an index that is out of range for an object.

This can happen when we try to call something out of range or when a loop tries to go beyond the length of an object.

Using len() can help prevent these exceptions as it can be used to prevent loops form iterating beyond the end of an object.

# Working with Lists

- Identifying the number of items in a list
  - To identify the number of items in a list, we can use the len() function.
  - The len() function returns the length of an object.
- Changing and modifying lists
  - Lists are mutable and as such, they can be changed and modified.
  - An expression like list[1] = new_value can be used to assign a new value to a list element.

# For Loops

# For Loops

A for loop is a count-controlled loop whereby the loop iterates a specific number of times.

For loops work very well with sequential data items as it iterates for every item in the sequence.

General Format:

```
for variable in [val1, val2, val3, etc.]:
    statements
```

# Strings

# Working with Strings in Python

- Strings are a specific data type in Python.
- There are many types of programs that perform operations on strings.
- Strings are essentially sequences in Python, so many tools can be used on them.
- To access an individual character in a string:
  - We can use indexing, as each character has an index starting at 0; just like lists.
  - Or we can use a for loop to iterate over the entire string.
    - For example, when we want to count the number appearances of a certain character in a string.

# Concatenating Strings

Concatenation is when we append one string to the end of another string.

We can use the + operator to produce a string that is the combination of the concatenated strings.

The operator += can also be used to concatenate strings. Here, the operand on the left side needs to already exist otherwise an exception will be raised.

Concatenating does not add spaces when the strings are combined.

# Strings are Immutable

- Strings are immutable (unchangeable) and once they are created, they cannot be changed.
- Concatenation doesn't actually change the existing string, but rather creates a new string and assigns the new string to the previously used variable.
- We cannot use the following expression because it will cause an exception.
    - string[index] = new_character

# String Functions

Strings in Python have many types of methods, divided into different types of operations.

Some functions test a string for specific characteristics. Generally, these will return a Boolean value, a True if a condition exists, a False if not.

Other functions work to modify strings and return a new value.

# String Testing Functions

| | |
|---|---|
| `isalnum()` | Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise. |
| `isalpha()` | Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise. |
| `isdigit()` | Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise. |
| `islower()` | Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise. |
| `isspace()` | Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`). |
| `isupper()` | Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise. |

# String Modification Methods

| | |
|---|---|
| `lower()` | Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged. |
| `lstrip()` | Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string. |
| `lstrip(char)` | The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed. |
| `rstrip()` | Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string. |
| `rstrip(char)` | The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed. |
| `strip()` | Returns a copy of the string with all leading and trailing whitespace characters removed. |
| `strip(char)` | Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed. |
| `upper()` | Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged. |

# Common String Methods

endswith(substring): checks if the string ends with the substring and returns T/F value.

startswith(substring): checks if the string starts with the substring and returns a T/F.

find(substring): searches for the substring within the string and returns the lowest index of the substring; if the substring is not contained in the string it returns a -1.

replace(substring, new_string): Returns a copy of the string where every occurrence of substring is replaced with new_string.

# Splitting a String

The split() function returns a list containing the words in a string.

By default, it uses the space as a separator, but other separators (period, slash, comma, etc.) can be used.

We can specify a different separator by passing it as an argument to the split() function.

# Formatting Output

# The format() function

- The format() function takes two arguments. The first is the string, float, or integer to be formatted, and the second argument is the formatting specifications.
- The format() specs include:
  - s: string
  - d: integer
  - f: float

# Dictionaries

# Dictionaries

A dictionary is an object that stores a collection of data.

- Each element consists of a *key* and a *value*.
  - This is often referred to as mapping a key to a value or a key-value pair.
  - A key must be an immutable object, or an object that cannot be changed.
- To retrieve a specific value from a dictionary, we use the key associated with it.
- The general format:
  - Dictionary = {key_1: value_1, key_2: value_2, etc.}
- Dictionaries can hold mixed data types as keys and values, but keys must be an immutable data type.

# Retrieving a Value from a Dictionary

- Elements are unsorted in a dictionary.
- The general format for retrieving a value from a dictionary is dictionary[key].
  - If the key is in the dictionary, the associated value is returned; if not, a KeyError exception is raised.
  - To help prevent KeyError exceptions, we can test whether a key is in a dictionary using an if/else statement and a not in operator.
- Dictionaries are mutable objects and new entries can be added.
  - To add a new key-value pair we can use:
    - Dictionary[key] = value
  - If the key exists in the dictionary, the value will be updated.
- To delete a key-value pair, we can use del dictionary[key] or dictionary.pop(key)
  - If the key is not in the dictionary, a KeyError exception is raised.

# Working with Dictionaries

- To create a dictionary we can use {} or the built in dict() function.
- The len() function can be used to obtain the number of elements within a dictionary.
- The .get() method can be used to get a value associated with the specified key from the dictionary; this is an alternative to the [] operator.
- The .keys() function can be used to retrieve all of the keys within a dictionary.
- The .values() function can be used to retrieve all of the values within a dictionary.
-