170D WOBC Module F

Python Programming I

US Government

Table of Contents

Chapter 1. Programming Overview

Objectives

- Understand programming’s role as sequential instructions.
- Explore types of programming languages.
- Compare syntax of English and Python.
- Learn the different approaches to following instructions, interactive vs. batch.
- Understand the difference between interpreted and compiled programming languages.
- Understand the difference between code and data.

Sequential Instruction

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until after teaching it to a computer, i.e., express it as an algorithm.
>
> — Donald Knuth, American Mathematical Monthly

In order to do anything in a consistent manner, people rely on instructions. Computers work in much the same way. Programming involves providing sequential instructions to the computer to solve a problem. Instructions can be simple or complex. The instructions can be followed in a sequential manner or the computer can move from one set of instructions to another according to need. For example, to prepare a meal, people often follow a recipe. How good that final meal tastes and how consistent it is depends on how closely the recipe is followed. In sports, we see examples of sequential instruction as well. A football coach might see a particular situation in a game and call a play for his players to advance towards the end zone. The play is a set of instructions designed for a particular goal. The instructions are given a name that all players on the team should be familiar with.

The recipe can be described as a *procedure*, a series of actions conducted in a certain order. The football play can be thought of as an *algorithm*, an ordered set of steps to solve a problem. An algorithm may consist of several procedures. Procedures and algorithms are combined to create a *process*, a series of actions to achieve a particular end. For example, there is a process that monitors spelling as someone types in a word processor. Additionally, to obtain a loan, an applicant follows a certain process with the bank to prove credit worthiness and secure funding.

Programming Languages

At the lowest level, computers do not understand human readable languages. The basic language of a computer is *binary code*, that is, 1 and 0. The 1 represents the state of something being on, while the 0 represents off. It would be tedious to communicate with a computer by writing or sending a series of 1s and 0s, so programmers use a programming language instead. The programming language is closer to a human readable format than binary. One or more tools translate the human readable programming language to binary. Binary may also be called *object code*.

Writing *software* or programs involves describing processes, procedures, and algorithms to a computer. Typically, this is done with a programming language. Eventually, the programming language is translated to binary instructions for the computer. A programming language can share varying degrees of similarity to a human language such as English. The programming language is a specific set of instructions designed to create procedures and algorithms. Referring to sports again, there is a particular set of terms used by football players to describe the plays. Dancers use specific terminology to describe a dance piece. Soldiers have a certain set of terms to describe movement and dangers on the battlefield. Each of these examples can be thought of as a programming language applicable to a particular situation. A programmer might describe these as a *domain specific language*. The language of football players is particular to the domain of a football game. Musical notation is another example of a domain specific language.

On the other hand, English, French, Mandarin, or Arabic are not languages designed for a specific purpose. These languages can be considered general purpose and although with some difficulty or nuance, a person can translate ideas from one of these languages to the other. There are domain specific and general purpose languages for computers as well. There are languages that solve a particular set of problems, such as Hypertext Markup Language (HTML) solves how to create the structure of a web page. There are also languages that can be applied generically to solve problems, such as C, Java, or Python.

Syntax

Military members learn a phonetic alphabet to relay letters and numbers over radio. The alphabet has very specific words and is pronounced in a particular way. For example, the letter P is Papa and pronounced with the accent on the second syllable, “pah-Pah”. More specifically, the number 3 is pronounced “tree”, not with a “th” sound.

These are examples of *syntax* rules. To communicate instructions clearly, the military has a specific syntax for speaking letters and numbers. In spoken languages, syntax is the set of rules that define how words and sentences should be structured. In computer languages, syntax follows the same principle. It defines how instructions are put together for the computer, the programmer, and other programmers. While most students learn basic syntax and grammar rules in elementary school, they also learn exceptions. Computers in general are much stricter when it comes to syntax than spoken language. Often, the first problem that a programmer discovers when learning is the *syntax error*. When entering instructions into the computer, their is a point where the computer confirms understanding of the syntax. If errors in syntax are present, the computer cannot correctly translate the statements into instructions. Spacing, case, and indentation are all matters of syntax and vary from one programming language to the next. In English, a student learns how many spaces to put after a period between sentences or how far to indent the beginning of a paragraph.

Python Basic Syntax

This section explains some fundamental syntax of the Python programming language.

Python is case sensitive. Therefore, the following two variables are different.

```
the_Person = "Him"
the_person = "Her"
```

Python statements do not need to end with a semi-colon, but one can be used to separate two statements if they are on the same line.

```
length = 10; width = 5
area = length * width
print(area)
```

All lines in a Python application must begin in the first column, unless the statement is in the body of a loop, conditional, function, or class definition.

- Python relies heavily on indention to determine the control flow structure of an application.
- As control structures are introduced, we will revisit the indentation rules.
- Indention of lines is typically a multiple of four spaces.

A long statement may be continued by placing the backslash (\) character at the end of the line.

- When used in this fashion it is often referred to as the line continuation character.
- Several examples of its use are shown below.

```
message = "This is a long string just to \
illustrate continuation across multiple lines"

result = 500 * 2 + \
400 * 3
```

Statements are automatically continued onto the following line(s) if the end of line is reached before the ending character for any of the following pairs of characters.

```
() [] {}
```

Below is an snippet of code using the `print()` function with multiple arguments.

```
data = "This is just a string of text"
print("This is the first argument",
      "This is the second argument",
      data)
```

While the arguments to the `print()` function are indicated through the use of commas, the additional whitespace used to separate the arguments being passed to the function is arbitrary and does not necessarily need to be consistent.

Any and all whitespace between the opening and closing parenthesis is permitted and is not bound by the indentation rules of Python.

Interactive vs. Batch

The amount of instructions when working with a programming language can be described as interactive or batched. For example, when driving to a new location, a person could use a GPS device with turn-by-turn directions. Each direction is supplied to the driver as needed, one direction at a time. If the driver makes a wrong turn or selects a different route, the following direction is appropriate for the new path. In this way, the driver is like a computer. The driver takes one direction at a time, interprets (translates) its meaning, and makes the appropriate action. Python can work in an interactive mode. This mode of computing uses a special program called a **REPL**. REPL stands for Read-Evaluate-Print-Loop. The Python REPL:

1. Reads what the user types
2. Evaluates the typed input as a Python *expression*
3. Prints out the results of that evaluation if any
4. Loops back to waiting for more input from the user

If instead, the driver were to purchase a map and plot out the entire route to their destination on it, that would describe a batch mode of programming. In Python, instead of typing out individual statements in a REPL, a programmer could use any text editor to create a file, usually with the extension *.py*, that lists all of the instructions in proper order. The programmer could then provide the file to a special program to be translated to machine instructions as a unit.

Compiled vs. Interpreted

Since the ultimate goal of a programming language is to take human-like language and translate it to something the computer (machine) understands, different programming languages define when to do the translation and with how much code at once.

At the United Nations, leaders often give speeches that are in their native language. These speeches are interpreted, statement by statement nearly simultaneously, by translators. The speech is not stored in the target listener's native language all at once.

When translating a book to another language the final copy of the target language is *compiled* together in one book. For example, you can read Harry Potter in the original English or find a copy translated to Spanish.

These choices are available to programming language designers as well. For example, Python is an *interpretted* language. The instructions are **stored** (whether in interactive or batch mode) in "native Python". In other words, the storage format of the program is human readable.

A programming language, like C, C++, or Java, on the other hand, is a *compiled* programming language. The original English-like source code is not the format stored for the computer. Instead, through a series of translation steps, the source file is converted and stored in its entirety to binary. When a user "runs" the program later, the program does not need to be translated. It is already in machine (binary) format.

Code and Data

A running program will have need to have more than just instructions. It will need to receive information from a user, from another computer, or from storage somewhere. For example, a word processor does not have all the documents ready-made in its code. Fundamentally, there is no difference between data and code at the binary level. Both instructions for the computer, also know as a program or code, and data are stored as bits in the computer's memory. From a logical labeling there is a difference.

Code is binary meant to provide instructions to the computer to perform some action. Data is binary as well but meant as input to or output from those instructions. In Python, a programmer may write some statements meant to provide instruction to the computer on what steps to take next. The programmer can also create a *variable* or *name* which is a logical reference to a certain set of bits in binary in a certain location in the computer's memory.

The good news is that the programmer does not have to worry about where those bits are nor how much space they will take up in memory. That is Python's job. However, the computer keeps track of which bits of binary are instructions and which bits are data for use in the program. In other languages, like C or Assembly, a programmer may be more concerned with the size and location of code and data when creating their program.

Exercises

Exercise 1

What is a syntax error?

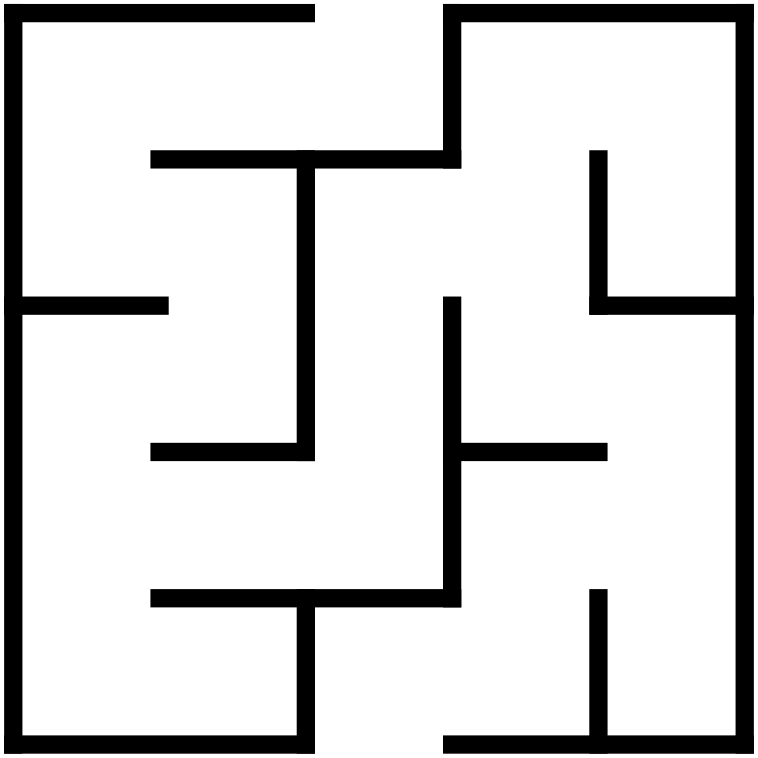
Exercise 2

Write out the instructions to play a game of musical chairs.

Exercise 3

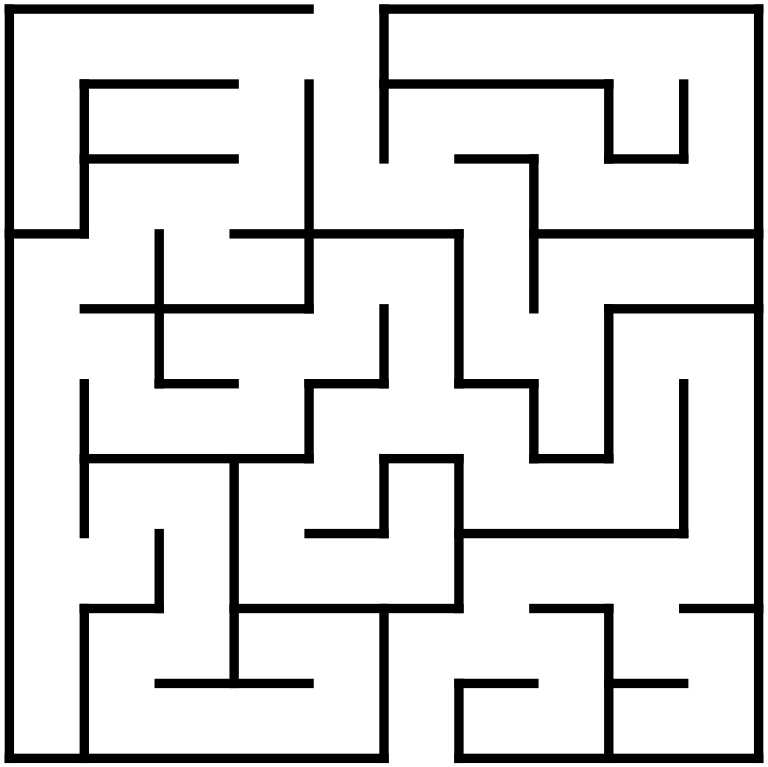
With the maze below, write directions to move through the maze using only the instructions: forward, left, and right. Write one instruction per line on your paper. Start at the bottom of the maze and move to the top exit. Directions are from the point of view as if a person were standing in the maze.

- *forward* moves one square
- *left* turns 90 degrees to the left, but does not move a square
- *right* turns 90 degrees to the right, but does not move a square



Exercise 4

Follow the instructions for the previous maze exercises with the maze below:



Is there a way reduce the number of lines in your set of instructions?

What if there were a new instruction called *repeat*?

repeat allows an instruction to be repeated a certain number of times

The following example moves forward 4 spaces:

```
repeat forward 4
```

This example turns the player around:

```
repeat left 2
```

Exercise 5

PPP (Pencil and Paper Programming)

Using only the available instruction set, guide another student to recreate a selected image in a 4-by-4 grid on graph paper.

- Some sample shapes can be found in the `programming_overview/` directory.

The following instructions are allowed:

- Move one square right
- Move one square left
- Move one square up
- Move one square down
- Fill-in the current square

Exercise 6

Repeat the previous exercise with a new image, however start with a simpler instruction set. Replace the long phrases with single words. The new instruction set becomes:

- right
- left
- up
- down
- fill

Exercise 7

This exercise builds on the previous. The new instruction set is now replaced with symbols.

- →
- ←
- ↑
- ↓
- ↙

These symbols represent move right, move left, move up, move down, and fill. Try a new image with these instructions.

Is the program shorter?

Is the program easier to follow?

Is the program faster to write?

Chapter 2. Running Python Programs

Objectives

- Understand the differences between python/python2/python3
- Running Python programs from the command line
- Use the Python REPL: Read-Evaluate-Print-Loop
- Understand literals
- Find and access Python documentation
- Access help()
- Describe Python keywords and objects with dir()
- Exit the Python shell
- Using comments
- Using preferred style for Python programs

Executing Python from the Command Line

Executing Python code can be done from the command line, Python's interactive interpreter, or an Integrated Development Environment (IDE). To execute a Python program from the command line, do the following.

1. First create a Python script with a text editor and name it with the standard `.py` file extension.
2. Once this file has been created, execute the file using one of the various python commands.

On a Linux system, the python command may refer to a version of Python 2 or Python 3.

- Python recommends the following convention to ensure that Python scripts can continue to be portable across Linux systems, regardless of the version of the Python interpreter.
  - `python2` will refer to some installed version of Python 2.
  - `python3` will refer to some installed version of Python 3.
  - `python` will refer to either Python2 or Python 3, depending on the platform.
- The above recommendation comes in the form of a Python Enhancement Proposal (PEP).
  - The specific PEP for the above recommendation is PEP 394.
  - More information about PEPs can be found at the following URL.

```
https://www.python.org/dev/peps/
```

The scripts provided in this course are written for Python 3 and as such rely on the `python3` command.

- Many of the scripts will include the following character sequence (known as the "shebang") as the first line of each source file.

```
#!/usr/bin/env python3
```

On a Linux system, the shebang provides the pathway to the Python interpreter. This could be a fixed path or directions to a symbolic link that describes the path to the Python interpreter.

A simple example script is shown below.

hello.py

```
#!/usr/bin/env python3
print("Hello World")
```

- The script would then be executed as follows:

```
$ python3 hello.py
Hello World
$
```

A more Linux-like approach is to first make the file executable and then simply execute it from the command line as shown below.

```
$ chmod u+x hello.py
$ ./hello.py
Hello World
$
```

- The `chmod u+x hello.py` is used to grant only the user (owner) of the file execution permissions.
- The `./hello.py` is used to execute the file.

This first program is offered merely to demonstrate the execution of a Python program from the command line.

- The `print` function sends data to the standard output.
  - In Python 3, parentheses are required for function arguments.
  - This is not the case in Python 2 versions where `print` was a statement as opposed to a function. This course presents more on statements and keywords later.

Executing Python from an Interactive Python Shell (REPL)

The Python interpreter, sometimes called an interactive Python shell, allows Python commands to be executed interactively.

- Interactively entering an expression will output the result of executing the expression without the need to call the `print()` function.
- This behavior is only available in the interactive shell and would not produce output if the same statement was run within a script.

Here is a small example session executing the Python interpreter in interactive mode.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Characters inside double quotes are a string")
Characters inside double quotes are a string
>>> print('Single quotes are also used to represent a string')
Single quotes are also used to represent a string
>>> 3 + 6
9
>>> result = 3 + 6
>>> value = 2 ** 10
>>> print("Passing multiple arguments", result, value)
Passing multiple arguments 9 1024
>>> print(value / result, value // result)
113.77777777777777 113
>>> # The pound sign makes this a comment
    #     The comment is here to indicate that the following
    #     defines a function that takes a parameter
    def some_function(param):
        print("The param is:", param)

>>> # The next statement calls the above function
    some_function(value)
The param is: 1024
>>> exit()
$
```

IDLE

Python provides a graphical tool named **IDLE** (Integrated Development and Learning Environment).

- Some of the features offered with IDLE are:
  - It is coded in 100% pure Python, using the `tkinter` GUI toolkit.
  - It is cross-platform: works on Windows, Unix, and Mac OS X.
  - It offers an interactive Python shell window with colorizing of code input, output, and error messages.
- As with the `python3` command discussed earlier, the IDLE command for Python 3 should be `idle3`.

IDLE can be started in various ways:

- Starting IDLE from the command line.
- Starting IDLE from the Applications menu.

Additional Editors and IDEs

The choice of a Python editor is largely a personal choice.

A list of common Python editors for various operating systems can be found at the following URL:

```
http://wiki.python.org/moin/PythonEditors
```

As we will see throughout the course, Python relies heavily on indention requirements of the source code to define blocks of code. Integrated Development Environments (IDEs) are usually configured to handle the spacing and indenting issues automatically for the developer.

Python as a Calculator

When a programmer first starts the Python interpreter, often they see a prompt like this:

```
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Recall that the Python interpreter is a REPL (Read-Evaluate-Print-Loop) program. Anything typed at this prompt is interpreted as a Python expression or statement. As in English grammar class, students learn the parts of speech and parts of a sentence, Python has language to describe the parts of programming code.

Expressions

Expressions in Python represent something. This could be something like a number or some text, also known as a string. Expressions are combinations of values and operators (such as the addition symbol or the subtraction symbol) that always evaluate to a single value. For example `2 + 2` is an expression and represents the value `4` in the interpreter. Expressions do not have to be just numbers. For example, `'Hello ' + 'world!'` becomes the single *literal* value `'Hello world!'`. (More in the Strings chapter.)

Statements

Python statements are made up of expressions that collectively do something. A statement is an action or a step in a program. Statements do not necessarily evaluate to a single value. For example `answer = 2 + 2` does not evaluate to a single value but instead evaluates the `2 + 2` part to `4` and then *names* that 4, `answer`. This is one type of statement called an assignment statement.

Literals

Literals are notations for constant values of some built-in data types. In the above examples, `4` is a literal as is `'Hello '`. These are not expressions that will be further evaluated to a single value. Literals are single values unto themselves.

It is not vital to programming to label the parts of Python code like diagramming English sentences, but knowledge of these terms will help in using Python documentation.

Python Documentation

There are various ways in which a Python programmer can get help from the Python documentation.

- A good starting point is one of the following URLs:
  - The most recent version can always be found here:

```
https://docs.python.org/3/
```

- The documentation for the specific version of Python being used can be found here:

```
https://www.python.org/doc/versions/
```

- The **Library Reference** and **Language Reference** links above are often useful to both new and seasoned Python developers.
- The **Python Setup and Usage** provides additional information for using Python on a specific operating system.

Getting Help

In addition to the online Python documentation, the interactive Python shell can provide additional help.

- Recall the Python shell can be started with either of the following.
  - The `python3` command for a text based environment.
  - The `idle3` command for a graphical environment.
- Once the Python shell is available, typing `help()` will start the Python help utility.
  - To see the math functions, type `math`.
  - To see the string methods, type `str`.
  - To see all documented topics, type `topics`.

Here is an example of using the help utility.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux

Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 3.8's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> math
```

- Upon typing math at the `help>` prompt, the documentation will be displayed as shown on the following page:

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

:
```

- Linux users might recognize the above view as a "man page", a form of documenting software.
  - The up, down, page up and page down keys on the keyboard can be used to scroll through the help screen shown above.
  - Typing the letter 'q' will quit out of the help screen above.
  - Typing `quit` at the `help>` prompt will exit the help utility back to the interactive Python shell prompt `>>>`
  - From there, typing `quit()` or `exit()` will exit the Python shell.
  - On Linux computers CTRL+D will also exit the shell. On Windows computers use CTRL+Z.

Alternatively, help can be obtained at the interactive Python prompt `>>>` by passing information

to the help function as shown below. `help('math') help('str')`

Comments

Many of the examples in this manual, even those entered at the interactive prompt, include comments.

Comments in Python start with the hash character, #, and extend to the end of the line.

A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when re-typing examples from this book.

Python Code Style

Although Python syntax requires certain capitalization of keywords, whitespace for indentation, and no semicolons as line endings, the style of Python syntax can vary from one programmer to the next. The Python Software Foundation provides a style guide, referred to as PEP 8, when formatting programs. Details of that style can be found online here:

```
https://www.python.org/dev/peps/pep-0008/
```

Python provides a tool called `pycodestyle` to check the style of a programmer's code against the PEP 8 guide. `pycodestyle` is not distributed with Python and would need to be installed separately. Every rule is not enforced by `pycodestyle`, but it can be used to develop a consistent coding style. In this course, students will be required to develop a consistent style for Python code.

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.
>
> — Guido van Rossum, https://www.python.org/dev/peps/pep-0008/

Using pycodestyle

Run pycodestyle from the command line with the script to check as the input. For example:

```
$ pycodestyle script_to_be_checked.py
```

Several options are available to change the way pycodestyle works. the `--first` option shows only the first occurrence of each style error.

Example

```
$ pycodestyle --first script_to_be_checked.py
```

The `--show-source` option can be useful when trying to find where the formatting error is precisely in the code.

Example

```
$ pycodestyle --show-source script_to_be_checked.py
```

Help is available from the command line.

Example

```
$ pycodestyle -h
```

Finding More about Values and Keywords

When working with the Python REPL, there may be times when a programmer needs more information about an expression, value, or *object*. The `dir()` function can be helpful in describing what *properties* a Python object such as a value may have. For example

```
>>> dir(4)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

That strange response from the Python interpreter is a *list* of properties and methods for the value 4. In other words, the interpreter provided a *directory* of all of 4's characteristics. From the listing, a programmer can infer that the value 4 can be **added**, **subtracted**, and compared to see if 4 is **greater than** another value. As more complex Python expressions and the use of Python "objects" are presented, `dir()` will become useful to explore new capabilities.

The Zen of Python

Part of learning a new language, whether human or programming, is learning a new philosophy and culture. Python programmers follow a general philosophy of "consistency over configuration". The basic ideas of this philosophy are found in the "Zen of Python". This is a Python script, also known as a *module*, that can be run from the Python shell. Access the Zen of Python with the following:

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
```

The output is omitted here. Try this on your own workstation.

An alternative way to run this *module* is to use the Linux command line and the `-m` option for the Python interpreter:

```
$ python3 -m this
```

Without the `-m`, the Python interpreter would not know to look for `this` in the Standard Library, a collection of Python code included when Python is installed on the computer.

```
$ python3 this
python3: can't open file 'this': [Errno 2] No such file or directory
```

Exercises

Exercise 1

What is PEP8? What is `pycodestyle`?

Exercise 2

What is the `shebang` and why should it be included in Python scripts?

Exercise 3

What is an expression?

Exercise 4

In a terminal try the following command:

```
python
```

What happens? What command is used to return control back to the command line?

Exercise 5

In a terminal try the following command:

```
python2
```

What happens?

Exercise 6

In a terminal try the following command:

```
python3
```

What version of Python is running?

What is the default version of Python installed on the system?

Exercise 7

Make all required changes to be able to execute the following program as `$ ./rpp_ex7.py`?

rpp_ex7.py

```
# A sample program
print("Hello World")
```

Exercise 8

Using `pycodestyle` and the file `bad_format.py` in the `running_python_programs/` directory, find and correct all PEP8 errors.

Chapter 3. Operators

Objectives

- Use Python operators
- Understand Python built-in math functions
- Use Variables
- Assign Values with =
- Perform Multiple Assignment
- Perform Chained Assignment
- Perform Augmented assignment

Operators and Numbers

Python supports three distinct numeric data types.

- Integers, Floating Point Numbers, and Complex Numbers

Python's numerical operations are shown below, sorted by ascending priority.

Table 1. Numerical Operations

Operation	Result
`x + y`	Sum of `x` and `y`
`x - y`	Difference of `x` and `y`
`x * y`	Product of `x` and `y`
`x / y`	Quotient of `x` and `y`
`x // y`	Floored quotient of `x` and `y`
`x % y`	Remainder of `x/y`
`-x`	`x` negated
`+x`	`x` unchanged
`abs(x)`	Absolute value (or magnitude) of `x`
`int(x)`	`x` converted to integer
`float(x)`	`x` converted to floating point
`divmod(x,y)`	The pair `(x // y, x % y)`
`pow(x, y)` or `x ** y`	`x` to the power `y`

In math class as children, students often learn that some mathematical operators have precedence over others. That precedence is shown in the preceding chart. In other words, in a sequence of operators in a statement, Python will calculate the value according to a set priority. For example, multiplication has priority over addition. So in the following:

```
>>> 2 + 3 * 3
11
```

Python will calculate `3 * 3` and arrive at `9` first. Python will then add `2` to `9` and give the result, `11`.

If a programmer truly meant for the addition operation to happen first, the programmer could use parentheses to indicate the addition has higher priority.

For example:

```
>>> (2 + 3) * 3
15
```

In this case, Python adds 2 to 3 first to get 5 and then multiplies that result by 3 to arrive at 15.

There is a simple mnemonic to remember the basic order of operations.

PE[MD][AS]

- Parentheses
- Exponents
- Multiplication and Division
- Addition and Subtraction

The brackets illustrate that with no parentheses to change the order, multiplication and division have equal priority and in most cases can be calculated as from left to right as they are encountered in the formula. The same is true for addition and subtraction.

Python Keywords

Python keywords are reserved words within the language. In algebra class students learn to name values with names like x, y, and π (pi). Likewise, a programmer can assign names to values in Python as well, however it would be wise to avoid any Python keyword.

- These words cannot be used as the names of variables or functions.
- They are listed here for reference.

Table 2. Python Keywords

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

The list above can be generated by executing:

- `help('keywords')` at the `>>>` prompt or
- `keywords` at the `help>` prompt.

Each keyword will be explained as new topics are introduced.

Naming Conventions

A Python **identifier** is a name that is used to identify any of the following:

- Variables
- Functions/Methods
- Classes
- Modules

Often Python programming documentation will use the term *name* to describe an identifier. An identifier must start with

- A letter of the alphabet or
- The underline `_` character.

This can be followed by any number of letters, digits and/or the `_` character.

- Identifier names cannot consist of any other characters.

Variable names and function names typically begin with a lowercase letter, while class names typically start with an uppercase letter. Remember that Python is a case-sensitive language. This means `mynumber`, `MyNumber`, and `Mynumber` are three different variables. Most Python programmers would prefer another option, `my_number`. The use of underscores between multi-word names is called **snake case**.

There are many different naming styles within the language. The previously mentioned PEP 8 describes these naming conventions in detail.

```
http://www.python.org/dev/peps/pep-0008/#naming-conventions
```

Assignment with =

A Python programmer can replace a value with a variable instead as a placeholder of sorts. For example, the following code would add two numbers together and display the results in the Python REPL.

```
>>> x = 3
>>> y = 2
>>> x + y
5
>>> x
3
>>> y
2
>>> z = x + y
>>> z
5
>>> y = 3.5
>>> z
5
>>> r = x + y
>>> r
6.5
```

Recall that the Python REPL is a **read-evaluate-print-loop**.

In the first statement, `x = 3`, the programmer **assigned** the value 3 (an integer) to the variable name x.

In the second statement, `y = 2`, the programmer performed a similar task. Now y **equals** 2.

It would better however to say that 2 **has been assigned** to y.

In the third statement, `x + y`, the programmer is not asking to see an output like `xy`, but instead, like in algebra class, to take the value assigned to `x` and add it to the value assigned to `y`.

This results in the output `5`.

At this point, there is no way to retrieve the result of `5` again. A later statement, `z = x + y`, saves the value of the calculation for later use with a variable named `z`. Now the programmer can

use z to recall the results of the calculation later in code.

Because the programmer is using the Python REPL, there is no need to use a Python keyword or function to output the results of a statement. For example, the programmer can simply type z and press ENTER to see the value assigned to z output or **printed** to the screen. In programs not designed to run in the Python REPL, a special Python function called `print()` will be necessary to tell Python to output the results.

In the code, the programmer **reassigned** the value 3.5 to the variable y. This section of the code is repeated below for reference.

```
>>> y = 3.5
>>> z
5
>>> r = x + y
>>> r
6.5
>>>
```

Any variable can be assigned to any value at any time in a program.

- It is the programmer's responsibility to ensure that a variable contains an appropriate value and data type to avoid unexpected results or errors.
  - ▶ Repeatedly re-using the same variable for different values and data types is **not** a good development practice.
- It is also the programmer's responsibility to utilize appropriate variable names.

The y variable was reassigned in the code and now refers to a new value.

Evaluation happens at the time of execution. Each line is executed as it is entered in the REPL.

Recall that previously in the code, z was assigned the results of the calculation `x + y`. Python evaluated `x + y` and assigned 5 to z.

If the programmer calls for z again by typing the name of the variable to be printed, we see 5. At the time of assignment to z, x referred to 3 and y referred to 3. In this example, Python does not re-evaluate `x + y` every time z is referenced.

The value assigned to y was changed later in the code and a new variable is chosen for the value of `x + y`, specifically r.

When the value of `r` is calculated, Python used the most recent value of `y` (now `3.5`) and the most recent value of `x` (`3`) and arrived at the answer, `6.5`.

It is important to understand that Python is "right-handed" in that Python evaluates the right hand side of an assignment statement first and then assigns that result to the variable on the left hand side.

For example this is correct:

```
x = 2 * 3 - (5 + 2)
```

The following is **incorrect** and will result in an error:

```
2 * 3 - (5 + 2) = x
```

- Chained Assignment

A quirk of this right-handedness in Python is that a programmer can use this to assign multiple variables to the same value at one time.

For example:

```
x = y = z = 2
```

This results in three separate variables x, y, and z referring to the value 2.

- Multiple Assignment

Another quirk of Python syntax is that multiple variables can be assigned to multiple values in the same statement.

For example:

```
x, y, z = 1, 2, 3
```

This results in three variables like the previous example, however x refers to 1, y refers to 2, and z refers to 3.

- Augmented assignment

The right-handedness of a Python assignment statement also gives a programmer another shortcut.

Often programmers need to perform a calculation on a value and refer to the new value with the same variable name. A counter in a game is a typical example.

Let's say a programmer wanted to count the number of times that an action had been performed, The programmer can make a variable called counter and assign it in this way:

```
counter = 0
```

This is called **initialization**. A variable in Python must be both declared (that is, have a name assigned) and initialized (that is, have a value assigned) at the time of creation.

For example, this is incorrect:

```
counter
```

A variable is declared but not initialized.

This is also incorrect:

```
= 0
```

A value is being assigned, but to what? Both steps happen when a variable is first created.

Now, suppose a programmer wants to increase the value of counter. The counter variable has already been declared and initialized with `counter = 0`.

Now a programmer could just reassign the counter variable to 1, such as:

```
counter = 1
```

While that works, a programmer might not know the exact value to assign to the counter, but might just want to increase it by one, or **increment** it.

The programmer could write:

```
counter = counter + 1
```

While this statement makes no sense in math class, this is a perfectly fine statement in Python. Remember that Python evaluates the right had side (RHS) first and then takes that result and assigns it to the variable on the left. In this case:

1. Python finds the current value of `counter`, that is 0.
2. Python then calculates `counter + 1` and arrives at the new value of 1.
3. Finally Python assigns the new value (1) to the variable `counter`. Now counter refers to 1.

Python has a syntax shortcut to obtain the same results. The programmer could type instead:

```
counter += 1
```

Admittedly, this is not a huge shortcut, but it is a common idiom in the Python language. The results are the same as if the programmer had typed `counter = counter + 1`. This is called an **augmented assignment**, which works with other operators as well.

Table 3. Augmented Assignment Operations

Operation	Result
`x += y`	Sum of `x` and `y` assigned to x
`x -= y`	Difference of `x` and `y` assigned to x
`x *= y`	Product of `x` and `y` assigned to x
`x /= y`	Quotient of `x` and `y` assigned to x
`x //= y`	Floored quotient of `x` and `y` assigned to x
`x %= y`	Remainder of `x/y` assigned to x
`x **= y`	`x` to the power `y` assigned to x

Exercises

For each of the following exercises, write a Python program to generate and print the solution to the problem.

Exercise 1

Three campers are going on a weekend trip. They have two 3 pound tents and an additional 27 pounds of food and gear. They want to evenly distribute all of the weight among each of their backpacks. How much weight will be in each backpack?

Exercise 2

A movie theater has 25 rows of seats with 20 seats in each row on the main level. Additionally the movie theater has a small balcony with 2 rows of 10 seats. How many seats are there in total?

Exercise 3

An Italian restaurant receives a shipment of 85 veal cutlets. If it takes 3 cutlets to make a dish, how many dishes can the restaurant make? How many cutlets will be left over?

Exercise 4

There is a group of 10 people who are ordering pizza. If each person gets a minimum of 3 slices and each pizza has 8 slices, how many pizzas should they order?

Exercise 5

A bricklayer stacks 4 rows of bricks, each row has 10 bricks. Later, 6 more bricks are stacked on top of each row. How many bricks are there in total?

Exercise 6

Mars has a diameter of approximately 9^4 kilometers. What is the diameter of Mars as a simple number?

Exercise 7

A day on Jupiter lasts about 10 hours. How many hours in 15 Jupiter days? How many days have passed on Earth in that time?

Chapter 4. Data Types

Objectives

- Explore basic data types of the Python language
- Use type() to find the data type a variable refers to
- Create sequences of characters as Python strings
- Get the length of a string with len()
- Use Boolean data types to represent True or False
- Understand the None type
- Cast data as one type or another
- Convert characters to their underlying values

Basic Python Data types

Python has many data types. There are builtin functions to convert from one type to another. If the source type cannot be converted to the target type, Python will show a `TypeError`. Broadly, these data types can be classified as Numbers, Strings, and Objects (more specifically, Sequence and Mapping types).

Number Data Types

Numbers can be integers (whole numbers with no decimal), floating point (those with decimal places), and complex. This chapter will look at integers and floating point numbers. In Python, integers are defined as type `int`. Floating point numbers, that is those numbers with decimal points and places, are defined as `float`.

The number `3` is an integer. The number `3.0` is a floating point number, as are `3.1`, `3.222`, and `3.01`.

Python3 does not limit the maximum value of an integer, it is unbounded. The size of the integer is limited to the size of the registers/memory allocated to the integer within the operating system. A floating point number is accurate to 15 decimal places.

Boolean Data Types

Additionally, there is another numeric data type known as `bool`. The `bool` data type is named after Boolean values. In short, a `bool` data type can hold only the values 0 or 1. This can be thought of as a way to express `True` (1) or `False` (0). For example, the `bool` value of 0 represents False.

Python supplies keywords to express True and False, these are unsurprisingly, `True` and `False`. The case is important. Both must start with a capital letter. Also make note that there are no quotation marks around `True` or `False`.

Another data type in Python is the `NoneType` which can be represented by using the `None` keyword. `None` is similar to *null* in other languages. In Python, `None` means nothing and will evaluate to `False` in a conditional statement. `None` is often used to check the result of some action or to ensure a variable has a value and is not nothing.

Although a programmer will usually not use these Boolean data types this way, a `True` can be thought of as an integer value of 1 as in the following code example.

```
>>> a = True
>>> a + 1
2
>>> a - 1
0
```

The following values are considered false:

- False
- Numeric zero (0 or 0.0)
- Empty collections (empty string, empty list, empty dictionary, empty set etc., more in later chapters)

Just about everything else is true.

Strings

A **string** is a sequence of characters (upper and lower case alphabetical characters, numbers, or special characters) surrounded by quotes. A string may also be referred to as a **string literal**. Recall that a literal is a basic value. This is raw data that cannot be changed once initialized. Python will recognize this data type as `str` for string. A Python expert would describe strings as an “immutable sequence of zero or more characters.” Immutable, in that strings cannot be changed once created. However, a variable referring to a string can be made to refer to a different string at another time in code.

Encoding

In Python, characters are **encoded** in Unicode by default. To a computer, every character is a number, this includes the letters and other characters typed on the screen. These numbering schemes indicate which number is the "code" behind a character. There is a code for each case of a character as well. There is even a code for each emoji you see in a program. Other coding schemes exist besides Unicode. Some older encoding standards include ASCII and EBCDIC.

A programmer can find the code behind a character by using the `ord()` function. The `ord()` function provides the **ordinal** (code) behind the character provided as input. For example, the Unicode code point for “A” is 65. For “a”, the number is 97. The code for the exclamation point is 33.

```
>>> ord('A')
65
>>> ord('a')
97
>>> ord('!')
33
```

The quotes around the letters and the exclamation point are important. They indicate that, in the example above, `A` is a string.

There is nothing special for a programmer to do with the Unicode value. The encoding is for the computer. The computer stores everything as numbers in memory, even the characters a user types. By keeping track that certain parts of memory are storing strings, Python knows that the numbers represent characters, not values to use in calculations. If a programmer initializes a string with a variable name, the variable name refers to a specific area in the computer’s memory. Python keeps track of the areas that have numbers (like integers and floats) and strings (which are really numbers too). In that way, Python keeps track of the data type of a value referred to by a variable and knows whether to use it as a number in a calculation or a number to represent a

character.

Python provides the `chr()` function to see which character is associated with a particular code.

For example:

```
>>> chr(97)
'a'
>>> chr(34)
'"'
>>> chr(33)
'!'
>>> chr(65)
'A'
```

- Literal string values may be enclosed in single or double quotes.
- Since strings are a sequence of characters, the built-in `len` function can be used to determine the number of characters the string contains.
- Literal strings can span multiple lines in several ways.
  - Using the line continuation character (`\`) as the last character.
  - Strings can be surrounded in a pair of matching triple-quotes: either double quotes(`"""`) or single quotes(`'''`).
- Literal strings may also be prefixed with a letter `r` or `R`.
  - These are referred to as raw strings and use different rules for backslash escape sequences.

If a programmer needs a special character in a string, the programmer may need to use an **escape sequence**. The following is a list of the escape sequences that can be used in literal strings to represent special characters.

Table 4. Escape Sequences

Sequence	Character/Meaning
`\newline`	Line continuation
`\\`	Backslash
`\'`	Single quote

Sequence	Character/Meaning
`\"`	Double quote
`\a`	ASCII Bell (BEL)
`\b`	Backspace
`\f`	Form feed
`\n`	Linefeed
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\ooo`	ASCII character (octal value `ooo`)
`\xhhh`	ASCII character (hex value `hhh`)
`\uxxxx`	Unicode Character with 16-bit hex value `xxxx`
`\Uxxxxxxxx`	Unicode Character with 32-bit hex value `xxxxxxxx`

The example below demonstrates the creation of literal strings using the various techniques described on the previous page.

string_literals.py

```
#!/usr/bin/env python3
print("This is a literal string", 'and so is this')
print('"Double quotes" inside of single quotes')
print("'Single quotes' inside of double quotes")
print("A double quote \" inside double quotes")
print(r"A double quote \" inside a raw literal string")
print("A as unicode: \x41")

spades = """Royal Straight Flush\
\U0001F0A1 \U0001F0AE \U0001F0AD \U0001F0AB \U0001F0AA
"""
print(spades)

diamonds = """Royal Straight Flush
\U0001F0C1 \U0001F0CE \U0001F0CD \U0001F0CB \U0001F0CA
"""
print(diamonds)
print("Emoticons 😎 😮 😳 😱 😴")

data = "~!@#$%^&*()_+"
print(data, "has", len(data), "characters")
```

Running the above program produces the following output:

```
$ python3 string_literals.py
This is a literal string and so is this
"Double quotes" inside of single quotes
'Single quotes' inside of double quotes
A double quote " inside double quotes
A double quote \" inside a raw literal string
A as unicode: A
Royal Straight Flush 🂡 🂢 🂣 🂤 🂥
Royal Straight Flush
🃏 🂢 🂣 🂤 🂥
Emoticons 😎 😮 😨 😱 😴💤
~!@#$%^&*()_+ has 13 characters
$
```

Note that while the use of Unicode within Python 3 is fully supported, a font that is capable of displaying the characters is necessary to produce the above output.

Dynamic Types

The following program emphasizes the dynamic type system of Python.

- In a dynamically typed language, a variable is simply a value bound to a name.
- The value of what is being referenced by the variable has a type like `int` or `float` or `str`, but the variable itself has no type.

datatypes.py

```
#!/usr/bin/env python3
x = 10
tab = " \t"

print(x, tab, type(x), tab, id(x))
y = x
print(y, tab, type(y), tab, id(y))
x = 25.7
print(x, tab, type(x), tab, id(x))
x = "Hello"
print(x, tab, type(x), tab, id(x))
```

The output from running the program above is shown below.

```
$ python3 datatypes.py
10      <class 'int'>           10943296
10      <class 'int'>           10943296
25.7    <class 'float'>         139761989022296
Hello   <class 'str'>           139761987959416
$
```

The built-in `type()` function returns the data type of the object that is referenced by a particular variable.

- Notice that the type of the reference is bound dynamically as the program is executed.

The built-in `id()` function returns a unique identifier of an object that is referenced by a particular variable. This is the address of a particular place in the computer's memory. It's not vital to know this address in normal Python programming but it can illustrate how Python stores

and accesses data in memory.

- There is only a single `int` object `10` being created and both the variables `x` and `y` reference the same object. This is evident in that they both produce the same id.
  - The benefit to having both variables refer to the same object is efficiency, it reduces the memory usage of the program.
  - If the value of one of the variables changes, a new object is created.

When working with integers, it may be convenient to encode literal values in bases other than decimal.

Base	Value	Decimal Value
Hexadecimal	`x = 0xff`	255
Octal	`y = 0o77`	63
Binary	`z = 0b111`	7

More information about numeric data types can be found in section 4.4 of the following URL:

`https://docs.python.org/3/library/stdtypes.html`

Conversion Functions

Often times, a programmer may need to treat a string as a numeric type or a number as a string within your code.

- Python provides several functions that allow these types of conversions to be performed.
- Some of the more common built-in conversion functions are:

`int()`	`str()`	`chr()`	`oct()`
`float()`	`ord()`	`hex()`	`bin()`

The `int()`, `float()`, and `str()` are methods called constructors that initialize an object as opposed to a function call.

Initialization of objects will be discussed later in the course in more detail. (More on this in the chapters on Classes and Objects)

conversions.py

```
#!/usr/bin/env python3
# conversions to an integer
data = "101"
number = int(data)
print(number, "plus 10 equals:", number + 10)
print("String of", data, "converted to various bases as int:")
fmt = "Base 10: {}\tBase 2: {}\tBase 8: {}\tBase 16: {}"
print(fmt.format(int(data), int(data, 2), int(data, 8), int(data, 16)))
print()
# conversions to a float
more_data = "23.45"
total = float(more_data) + float(data)
print(more_data, "+", data, "=", total)
print()
print(number, "as string in the following bases:")
fmt = "Binary: {}\tOctal: {}\tHex: {}"
print(fmt.format(bin(number), oct(number), hex(number)))

print('ord("A") =', ord("A"), '     chr(66) =', chr(66))
```

The output from running the example on the previous page is shown below.

```
$ python3 conversions.py
101 plus 10 equals: 111
String of 101 converted to various bases as int:
Base 10: 101    Base 2: 5   Base 8: 65  Base 16: 257

23.45 + 101 = 124.45

101 as string in the following bases:
Binary: 0b1100101   Octal: 0o145    Hex: 0x65
ord("A") = 65      chr(66) = B
$
```

If the arguments passed to the conversion functions are not of the proper form, an exception will be generated as shown below.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> result = int("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
>>> result = int("123", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 2: '123'
>>> print(float("987"))     # While this works
987.0
>>> print(int(98.76))       # And this works also
98
>>> print(int("98.76"))     # This one does not work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.76'
>>> exit()
$
```

Exercises

Exercise 1

Write a program that calculates the length of the following string:

'Python is not difficult to learn.'

Exercise 2

Write a program that calculates the product of two integers and print the results as a float.

Exercise 3

Looking at the values in the table below, what data type would Python interpret them to be?

Value	Data type
'dog'	
3 > 2	
1.273	
-1.273	
ord('B')	
4 ** 2	
'f' * 2	
'cat' + '2'	
True	

Exercise 4

Write the code to output the type for each of the values in the previous table.

Exercise 5

Write the code to store `2` and `12.7` as separate variables and return the calculation as a float.

'2' + 12.7

Exercise 6

What character is represented by each of the following Unicode values (code points)? Use a Python function to obtain the answers.

Value	Unicode Character
72	
33	
126	
0x25	
174	
131 % 66	

Exercise 7

Answer the following questions.

1. When should a programmer use ''' or """ (triple quotes)?
2. What are the three types of numbers in Python?
3. What data type is returned by `10 / 3`?
4. What data type is returned by `10 // 3`?
5. Are `'27'` and `27` the same data type? If not, why? What data types are they?
6. What data type is `(3 / 2) < 1`?
7. What function does a programmer use to convert `3 * 1.2` to an integer?

Exercise 8

Write a program to print the square root of 25. (Hint: A programmer could think of the square root as raising a value to 1/2.)

Chapter 5. Basic I/O

Objectives

- Gain greater knowledge of `print()`
- Make use of `input()` function
- Write and understand f-strings
- Read and understand `.format()`-style strings
- Read and understand %-style formatting

Simple Output

The `print()` function has been demonstrated in several forms in previous examples.

- This function takes zero or more arguments, separated by commas, and outputs the data to the display.
- By default, the following two things occur.
  - When there is more than one argument, each of the arguments will be sent to the display separated by a single space.
  - A newline is sent to the display as the last thing it does.

The programmer can alter the default behavior of the `print()` function by providing values for the following optional named arguments.

- `sep`
  - A string of zero or more characters to be used as the separator rather than the default space.
- `end`
  - A string of zero or more characters to be printed last rather than the default newline.
- `file`
  - This is the data stream that is written to that defaults to the system console if not specified
  - Use of this named argument will be shown in a later chapter.

While a more detailed explanation of named arguments will be discussed in a later chapter, the example below demonstrates their use with the `print()` function.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello", 123, "Goodbye", 5 + 9, sep=":", end="###")
Hello:123:Goodbye:14###>>> exit()
$
```

Simple Input

The `input()` function provides a mechanism to obtain user input at run-time from the keyboard.

- The function accepts an optional prompt string as an argument.
- If provided, the prompt string is printed to the system console exactly as given.
- No spaces or other formatting characters are added.
- The `input()` function automatically removes the trailing newline from the keyboard input prior to returning the entered data.
- The return value of the `input()` function is always of type string.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> response = input("Please enter some text:")
Please enter some text:*Hello*
>>> response = input("Please enter some text:\n")
Please enter some text:
Goodbye
>>> print("The last response is", response, sep=":")
The last response is:Goodbye
>>> exit()
$
```

Note that even when a number is supplied as input, it is always returned as a string as seen in the following example.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> response = input("Please enter a number: ")
Please enter a number: 456
>>> print("The response of", response, "is of type", type(response))
The response of 456 is of type <class 'str'>
>>> exit()
$
```

String Formatting Methods

Sometimes a program requires more than just basic output. Python has several ways to format the output rather than just printing space separated values.

The newest and easiest way to format a string for output in Python uses formatted string literals or "f-strings". Beginning with Python 3.6, f-strings are available for output formatting.

To use a formatted string literal, begin the string with f or F before the opening quotation mark. This can be used in front of single, double, or triple quotation marks. Inside the string, the programmer adds braces such as { and }. The variable that should be output is placed between the braces.

```
>>> name = 'Pat'
>>> flavor = 'vanilla'
>>> f'My name is {name} and I like {flavor} ice cream.'
'My name is Pat and I like vanilla ice cream.'
```

Because f-strings are evaluated when the program is run, a programmer can put any valid Python expression in the them. For example:

```
>>> f"Two times two is {2 * 2}"
'Two times two is 4'
```

A programmer can define and merge f-strings to create a multi-line f-string. For example:

```
>>> name = 'Chris'
>>> job = 'tech person'
>>> prog_lang = 'Python'
>>> content = (
...     f"The programmer is {name}. "
...     f"{name} is a {job}. "
...     f"{name} programs with {prog_lang}."
)
>>> content
'The programmer is Chris. Chris is a tech person. Chris programs with
Python.'
```

Notice that the `{name}` placeholder, or *field* can be repeated. Python evaluates the fields in the braces as Python expressions and looks for variables to match. The order of the fields does not matter and as stated earlier, a field can contain any valid Python expression. It is important to include the `f` or `F` in front of each line.

- F-strings are described in full detail at

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

Format Specification Mini-Language

When using f-strings the programmer can place curly braces in a literal string. The curly braces can contain a Python expression. Additionally, the programmer can specify additional formatting details for the result of the Python expression. To specify those details, the programmer uses the Python Format Specification Mini-Language.

A format specifier can be appended to the field after a colon `:` and inside the curly braces. Backslashes are not allowed within the formatting field and will cause a syntax error. Typically, a programmer does not need to use a format specifier if the field will only be replaced with text. If the programmer wishes to improve the formatting of numeric variables for printing, then a format specifier within the field is helpful.

Table 5. Format Specification and Meaning

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
None	The same as 's'.
'b'	Binary format. The number is displayed as base 2.
'c'	Character. Converts the integer to a Unicode character.
'd'	Decimal Integer. This converts the number to base 10.
'x' or 'X'	Hex format. Display the number in base 16. The case of the hex number will match the case of the specifier.
'f'	Fixed-point number. Displays the number with a decimal point. Default precision is 6.
'%'	Percentage. Multiplies the number by 100 and displays in fixed ('f') format with a percent sign.

There are many other format specifiers, but those shown here are commonly used.

String formats will be left aligned by default, numeric formats will be right aligned. This can be controlled by using `<` to left align, `^` to center, and `>` to right align.

In the following example, the variable `interest_rate` is created which has a floating point value.

```
interest_rate = 5.789
print(f'{interest_rate:.2f}')
```

When the code is run, Python will output `5.79`. The field for `interest_rate` in the `print` statement has a format specifier after the variable name. The `.2f` may be more easily understood from the right end. The 'f' indicates the programmer would like the output to be fixed-point. The '.2' indicates that the programmer would like two places after the decimal point, in other words, two places of precision. In the output, Python has rounded `5.789` to `5.79`. The rounding is for output only and does not change the underlying value of the `interest_rate` variable.

The following code example has four variables, each referring to an integer. The variables are labeled `ip_add_octet1`, `ip_add_octet2`, `ip_add_octet3`, and `ip_add_octet4`. Each one represents on octet of an IP address.

```
ip_add_octet1 = 192
ip_add_octet2 = 168
ip_add_octet3 = 0
ip_add_octet4 = 1

print(f'{ip_add_octet1}.{ip_add_octet2}.{ip_add_octet3}.{ip_add_octet4}')

print(f'{ip_add_octet1:b}')

print(f'{ip_add_octet1:x}')
```

The code will produce the following output:

```
192.168.0.1
11000000
c0
```

The variable `ip_add_octet1` refers to the integer value `192`. In the code, the variable, along with the three other octets, is printed as part of a normal IP address with literal periods '.' between each. Then the variable `ip_add_octet1` is printed in binary format (base 2). It is important to know that f-strings are only changing the output. The stored value for the variable is not changed. Finally, the last `print` statement prints the first octet as a hexadecimal value.

Additionally, a value indicating the minimum width can be specified before the format. For example, `5s` would indicate a string with a minimum width of 5 characters. If the string being inserted is fewer than 5 characters, it will be padded with spaces. If the string being inserted is more than 5 characters, the additional space required will be provided. Another example would be to insert commas for larger numbers, such as `10,d` or `7,.2f`.

```
name = 'Mel'
job_title = 'head honcho'
value = 1234567.0987
print(f'name: {name:>10s}.')
print(f'title: {job_title:3s}.')
print(f'value: {value:7,.3f}')
```

The code will produce the following output:

```
name:        Mel.
title: head honcho.
value: 1,234,567.099
```

The str.format() Method

Earlier, starting with version 2.6 of Python, an alternative string formatting option was introduced. Like f-strings, replacement fields are indicated with curly braces '{}'. However, the syntax of referencing the variables to be printed was different.

- The `format()` method is described in full detail at

https://docs.python.org/3/library/string.html#formatstrings

```
output = 'This will print the first octet as {}'.format(ip_add_octet1)

print(output)
```

The preceding code produces the following output:

```
This will print the first octet as 192
```

Much like f-strings, the str.format() method uses curly braces in a template string to mark where the output should be replaced. `format()` is a method to be used with strings. The syntax is to create a string literal with fields marked with curly braces '{}' where the programmer wants the output to appear.

With no other indications, the order of the curly braces matter, as in the following.

```
pet = 'dog'
name = 'Fido'
owner = 'Susan'

output = '{} has a {} named {}.'.format(owner, pet, name)
print(output)
```

This will produce the following output:

```
Susan has a dog named Fido.
```

If the order of the variables in the `format()` method changes, so will the output. As in:

```
pet = 'dog'
name = 'Fido'
owner = 'Susan'

output = '{} has a {} named {}.'.format(name, pet, owner)
print(output)
```

This produces the following:

```
Fido has a dog named Susan.
```

Notice the new order of variables in the `format()` method. Although it is possible that Fido has a dog named Susan, this is probably not what the programmer wanted. Each of the variables has an index within the method. For example, if this is the method: `str.format(name, pet, owner)`, then `name` is in position 0, `pet` in position 1, and `owner` in position 2. So the programmer could write the following to make the output appear like the first example.

```
pet = 'dog'
name = 'Fido'
owner = 'Susan'

output = '{2} has a {1} named {0}.'.format(name, pet, owner)
print(output)
```

The numbers indicate the index of the variable in the format method. This will produce output as in the first example.

```
Susan has a dog named Fido.
```

The programmer could use names instead of indices much like what has been shown with f-

strings. For example:

```
output = '{owner} has a {pet} named {name}.'.format(name='Fido',
pet='dog', owner='Susan')

print(output)
```

This produces the same output but now the values for each field have been supplied as *named arguments* in the `format()` method. Used in this way, the order of the fields no longer matters. The same format specifiers used in f-strings will work with the `format()` method.

```
ip_add_octet1 = 192
output = 'This will print the first octet as hex with uppercase letters
{:X}'.format(ip_add_octet1)

print(output)
```

This preceding code produces:

```
This will print the first octet as hex with uppercase letters C0
```

The colon ':' separates the index or name of the field from the format specifier.

Original Formatting with %

Although this style of formatting strings is no longer recommended, a Python programmer should be familiar with using *%-formatting*. Frequently, a developer will come across older Python code that will need to be updated, an understanding of the original Python string formatting syntax will be helpful.

Strings have a built-in operation using the '%' operator which is used to format strings. For example:

```
name = 'Pat'
age = 42
salary = 1.737
output = 'Hello, %s. You are %s. You make %.2f' % (name, age, salary)
print(output)
```

Instead of the usual curly braces as seen in earlier examples, this style of string formatting using '%' followed by a format specifier to indicate where text substitution should take place. The first `%s` will be replaced with the string 'Pat' and the second `%s` will be replaced with 42. Finally, the `%.2f` will be replaced with 1.74, the value 1.737 with a precision of 2 places after the decimal point. The format specifiers are the same ones that are available in later styles of string formatting.

- Using the `%` operator for string formatting is described in full detail at

https://docs.python.org/3/library/stdtypes.html#printf-style-stringformatting

Exercises

Exercise 1

Write a program to display "I like Python Programming" as `I*-*like*-*Python*-*Programming` using the print() function.

Exercise 2

Convert the decimal number 42 to hexadecimal using print() function formatting.

Exercise 3

Display the number 3.14159265 with 5 decimal places using print().

Exercise 4

Print the numbers 1, 2, and 3 on separate lines with one print() statement.

Exercise 5

Using the following data:

```
quantity = 4
item = 'donuts'
price = 1.25
```

Write a program to calculate the total cost and output the following:

```
Thank you for purchasing 4 donuts at $1.25 each for a total of $5.00!
```

Exercise 6

Write a program to accept three numbers from the user and print the average. Output should be similar to:

```
Enter first number: 3
Enter second number: 4
Enter third number: 5
The average is 4.00
```

Exercise 7

Write a program that asks a user for their name and their age. Print out the year that they will turn 67 years old.

Exercise 8

Write a program to print the following string including the quotation marks:

```
"Ain't" ain't a word and you ain't going to use it.
```

Chapter 6. Lab 01

Objectives

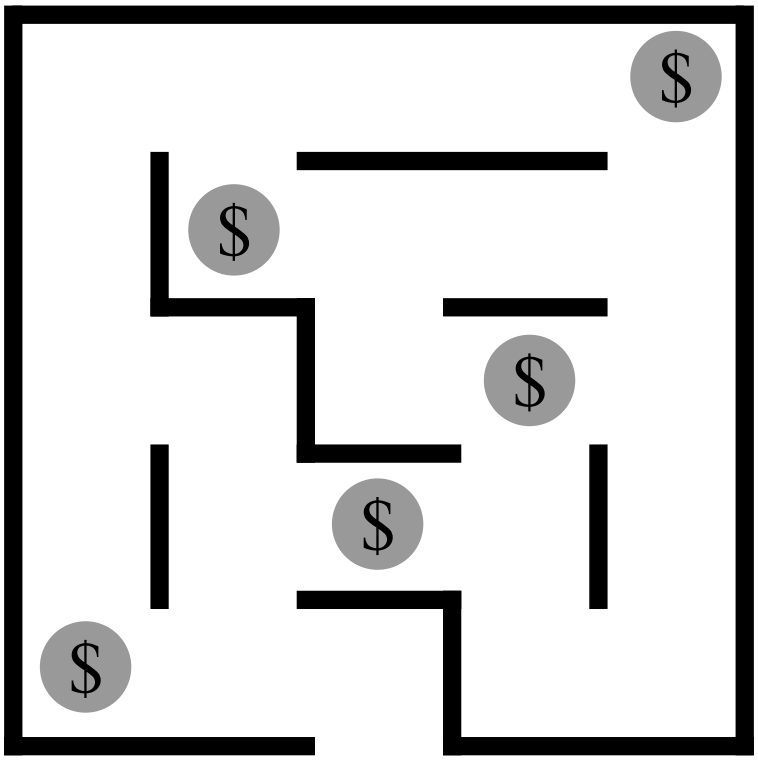
- Gain a better understanding of previously covered concepts of the Python programming language through the completion of practical exercises.

Exercises

Exercise 1

With the maze below, write directions to move through the maze, picking up each coin as you go, using only the instructions: forward, left, and right.

- Write one instruction per line on your paper.
- Start at the entry to the maze and move until each coin is picked up before returning the the entry point.
- Moving into a square that has a coin will collect that coin.
- Each coin can be collected only once, though its path can be crossed multiple times.
- Directions are from the point of view as if a person were standing in the maze.
  - *forward* moves one square
  - *left* turns 90 degrees to the left, but does not move a square
  - *right* turns 90 degrees to the right, but does not move a square



Exercise 2

With the name tags shown below, write directions to move each of the name tags so that they are in alphabetical order.

- Use only the directions left and right.
- Write one instruction per line on your paper.
- Each instruction should include the name tag and the direction, such as: Andrew right
- Each instruction will only move the given name tag one location to the left or right.
  - ▶ *Trevor left* moves Trevor between Yvonne and Declan.

Exercise 3

Use the interactive `help()` function to find additional information on 4 topics from the list of available `topics`.

- Some recommended topics include `str`, `int`, `float`, `bool`, `print`, and `input`.

Exercise 4

Use the interactive Python shell to assign a literal string for a first name in one variable and another literal string for the last name. Print the information in the two variables as a full name. Comments should be used before each variable assignment to indicate what the code does.

Exercise 5

Repeat the previous exercise using IDLE instead of the interactive Python shell

Exercise 6

Place the code used in the previous two exercises into a python file, with a name of your choosing. The file name should be indicative as to what the program does. Run the program from the command line.

Exercise 7

Since IDLE can be used as an editor, use it to open and edit the file created in the previous

exercise.

The file should be edited such that the last name is printed before the first name, separated by a comma. Rather than save over the original file, Save this version of the file under a new name and execute (run) it.

Exercise 8

A person receives a dime, 2 nickels, 2 quarters, and 4 pennies in change from a purchase. Write a program to calculate how much money was received in change.

Exercise 9

A video streaming service is being offered for $15 a month with a 10% discount for buying a years worth in advance. Write a program that answers the following questions:

- What is the yearly price for someone paying per month and the discounted yearly rate?
- How much is saved in a year by purchasing the yearly membership over the monthly membership?

Exercise 10

What is the difference in height (in inches) of one person who stands 7'2" and another that stands 5'8"?

Exercise 11

Create four literal strings, with each one holding the Unicode characters for the 13 playing cards in a suit: Ace, Two, … Queen, King.

Print the four strings to output a deck of cards.

Exercise 12

Print out the number 129 in binary, octal, and hexadecimal. Then define variables to hold the number 129 supplied as a binary literal, octal literal, and a hexadecimal literal value.

Exercise 13

Given the following statement: `letter = "A"`. Use the `ord()` and `char()` functions to print out the letter `B`.

Exercise 14

Store the literal string of *Hello* in as many ways as you can think of. Each representation should be stored in its own variable.

Exercise 15

Determine how many different string objects are stored in the variables below.

```
farewell_a = "Goodbye"
farewell_b = "Goodbye"
farewell_c = 'Goodbye'
farewell_d = "Good" + "bye"
farewell_e = "good" + 'bye'
farewell_f = "goodbye"
farewell_g = 'adios,' + 'dag,' + 'さようなら,' + 'Auf Wiedersehen,' +
'later'
farewell_h = str(farewell_g)
```

Exercise 16

Print the numbers 1 through 10, inclusive, as comma separated values all on the same line.

Exercise 17

Using a single `print` statement: Print the numbers 1 through 10, inclusive, with each number on its own line in the output.

Exercise 18

Write an application that prompts the user to enter a name three times and stores each name in its own variable.

After collecting the names, print all of the names on a single line.

Exercise 19

Rewrite the previous application so that it also prints, on a separate line, the total number of characters in all of the names supplied.

Exercise 20

Rewrite the previous application once again so that the output: Prints each name and the number of characters in the name. The name should be first and the length second. The name and length should be right justified with a minimum width of 15.

Exercise 21

A student wishes to be able to determine their grade average after taking 5 tests. write an application that allows the user to input all 5 test scores and print out the average.

Chapter 7. Review

Objectives

- Running and Writing Python Programs
- Operators
- Data Types
- Basic I/O

Running Python Programs

When writing Python code it can be written and executed interactively as each statement is typed, or all written to a file and then running the batch of statements at once.

Interactive programming can be done using the Python Interactive Shell and its Read-Evaluate-Print-Loop(REPL) capabilities.

- This process usually involves the following steps:
  - *R*ead what the developer types.
  - *E*valuate the typed input as a Python expression.
  - *P*rint out any results of that evaluation.
  - *L*oop back to waiting for more input from the developer.

Batch programming can be done using a text editor that allows multiple Python statements to be written and saved to a file.

- This process usually involves the following steps:
  - Create a Python script/program with a text editor and save the file with the standard .py extension.
  - Execute the file from the command line either as an executable file or supplying the filename to the Python interpreter.

The next several examples will calculate the volume of a box for a given length, width, and height.

- The first approach will utilize interactive programming.
- The second approach will utilize batch programming.

The interactive shell will be used to type in the code interactively in this example.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # Conversion Factor
>>> inches_per_foot = 12
>>>
>>> # Length of box (feet)
>>> length = 2
>>>
>>> # Width of box (inches)
>>> width = 3
>>>
>>> # Height of box (inches)
>>> height = 5
>>>
>>> print("Volume of box is:", length * inches_per_foot * width * height,
"cubic inches")
Volume of box is: 360 cubic inches
>>> exit()
$
```

- Comments were used to help distinguish which measurements were in inches and which were in feet.

The batch approach is demonstrated in the next example that places all of the above code in a `.py` file.

volumes.py

```
#!/usr/bin/env python3

# Conversion Factor
inches_per_foot = 12

# Length of box (feet)
length = 2

# Width of box (inches)
width = 3

# Height of box (inches)
height = 5

print("Volume of box is:", length * inches_per_foot * width * height,
"cubic inches")
```

One way to run the above program is to run the python interpreter with the name of the program from the command line.

```
$ python3 volumes.py
Volume of box is: 360 cubic inches
$
```

Another way is to make the file executable and execute it as a bash script.

```
$chmod u+x volumes.py
$ ./volumes.py
Volume of box is: 360 cubic inches
$
```

Writing Python Programs

Style guides provide a consistent approach to writing software. It can also add to the readability of the code. Python provides a standard style guide known as PEP8, and a tool to check compliance with the guide called pycodestyle.

The example below shows the results of testing the previous example, `volumes.py`, using the pycodestyle tool.

```
$ pycodestyle volumes.py
volumes.py:15:80: E501 line too long (85 > 79 characters)
$
```

The output indicates that line # 15 in the file has too many characters (at most 79)

- The examples below show several different approaches to changes that can be made to make it PEP8 complaint.

volumes_fix01.py

```
#!/usr/bin/env python3

# Conversion Factor
inches_per_foot = 12

# Length of box (feet)
length = 2

# Width of box (inches)
width = 3

# Height of box (inches)
height = 5

print("Volume of box is:", length * inches_per_foot * width * height,
      "cubic inches")
```

volumes_fix02.py

```
#!/usr/bin/env python3

# Conversion Factor
inches_per_foot = 12

# Length of box (feet)
length = 2

# Width of box (inches)
width = 3

# Height of box (inches)
height = 5

volume = length * inches_per_foot * width * height

print("Volume of box is:", volume, "cubic inches")
```

The following example would be an example of doing the wrong thing to make it PEP8 compliant.

volumes_bad_fix.py

```
#!/usr/bin/env python3

# Conversion Factor
ipf = 12

# Length of box (feet)
l = 2

# Width of box (inches)
w = 3

# Height of box (inches)
h = 5

print("Volume of box is:", l * ipf * w * h, "cubic inches")
```

- Shortening the variable names to make the line have fewer characters on it leads to less readable code.
  - While it fixes the problem of too many characters, it introduces bad practices to accomplish it.
  - `pycodestyle` will now complain about the variable `l` being too ambiguous because in certain fonts it is hard to determine if it is the lower case letter "L", "I", or the number "1".

Operators

When performing numerical operation on data, it is important to understand to order of operations for each of the operators.

- The PE[MD][AS] mnemonic can be used to help remember the basic order of operations.
  - *P*arnetheses
  - *E*xponents
  - *M*ultipliation and *D*ivision
  - *A*ddition and *S*ubtraction

When evaluating the following expression:

```
result = 10 / 2 - 5 * 3 + 25
```

- Python first evaluates the 10 / 2 as if the expression was now:

```
result = 5.0 - 5 * 3 + 25
```

- Notice how the divison operation resulted in a float as a value
  - ▶ It then evaluates the 5 * 3 as if the expression was now:

```
result = 5.0 - 15 + 25
```

- It then evaluates the 5 - 15 as if the expression was now:

```
result = -10.0 + 25
```

- It then evaluates the -10 + 25 as if the expression was now:

```
result = 15.0
```

- The last step in the process is to store the final value in the variable to the left of the = sign as shown above.

Often, it is desired to swap the values stored in two variables.

swapping.py

```
#!/usr/bin/env python3
x = "Hello"
y = "Goodbye"
print("x =", x, "y =", y)

temp = x
x = y
y = temp
print("x =", x, "y =", y)
```

```
$ python3 swapping.py
x = Hello y = Goodbye
x = Goodbye y = Hello
$
```

Augmented assignments in Python provide a more Pythonic way of performing the above.

pythonic_swapping.py

```
#!/usr/bin/env python3
x = "Hello"
y = "Goodbye"
print("x =", x, "y =", y)

# Augmented assignment to swap values in variables
x, y = y, x
print("x =", x, "y =", y)
```

```
$ python3 pythonic_swapping.py
x = Hello y = Goodbye
x = Goodbye y = Hello
$
```

Data Types

Two common numeric data types in Python are `int` and `float`.

- Literal numbers with no decimal point are of type `int`
  - Some examples are `5`, `-9`, `256`, `100000`, `0x23`
- Literal numbers with a decimal point are of type `float`.
  - Some examples are `5.7`, `-67.0034`, `99.`

The `type()` function can be used to determine the data type of any value in Python.

```
$ python3
Python 3.8.2 (default, Mar 18 2020, 21:14:39)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print(type(9), type(9.0), type(8/4), type(1/3))
<class 'int'> <class 'float'> <class 'float'> <class 'float'>
>>>
>>> x = "123"
>>>
>>> # Converting a string to an int
>>> y = int(x)
>>>
>>> print(type(x), type(y))
<class 'str'> <class 'int'>
>>> exit()
```

A string in Python is represented as an object of type `str`.

A string is an immutable sequence of zero or more characters.

- Literal strings are created by placing quotes around characters as data in your code.

The example below demonstrates the various ways literal strings can be created and some functions that convert numbers to strings.

reviewing_strings.py

```
#!/usr/bin/env python3
# Literal strings
greeting = "This is a string in double quotes"
statement = 'This is also a string, but in single quotes'
quotes_in_quotes = "Placing a double quote (\") inside of double quotes"
no_escaping = 'Placing a double quote (") inside single quotes'
backslashes = "A backslash \\ needs to be escaped to be a backslash"
raw_strings = r"This backslash \ needs no escaping"

multiline_strings = """Multiline strings can be created with 3 sets of
quotes
and everything between tbem make up a single  string, including the
newlines
in this string"""

escaping_newlines = '''Escaping newlines is done with a \\ \
at the end of a line.'''

# Print all of the strings with a new line between each as separator
print(greeting, statement, quotes_in_quotes, no_escaping, backslashes,
      raw_strings, multiline_strings, escaping_newlines, sep="\n\n")

print()

# Converting numbers to strings
letter_a = chr(65) + chr(97)
binary_101 = bin(101)
octal_101 = oct(101)
hexadecimal_101 = hex(101)

# Print strings obtained from functions (as opposed to literal strings)
print(letter_a, binary_101, octal_101, hexadecimal_101)
```

```
$ python3 reviewing_strings.py
This is a string in double quotes

This is also a string, but in single quotes

Placing a double quote (") inside of double quotes

Placing a double quote (") inside single quotes

A backslash \ needs to be escaped to be a backslash

This backslash \ needs no escaping

Multi-line strings can be created with 3 sets of quotes
and everything between them make up a single  string, including the
newlines
in this string

Escaping newlines is done with a at the end of a line.

Aa 0b1100101 0o145 0x65
$
```

Reviewing Basic I/O

The `print()` function in Python is used to print zero or more arguments to the display.

- By default there are several things it does automatically:
  - When there is more than one argument passed to the function, each argument will be separated by a single space.
  - Regardless of the number of arguments (even zero), the function will always send a newline as the last thing to the display

A programmer can alter the above default behavior of the `print()` function by providing the following optional named arguments.

- `sep`
  - A string of zero or more characters that will be used as the separator instead of the default space.
- `end`
  - A string of zero or more characters that will be appended at the end instead of the default newline.

print_stuff.py

```
#!/usr/bin/env python3
# Literal strings
sunday = "Sunday"
monday = "Monday"
tuesday = "Tuesday"
wednesday = "Wednesday"
thursday = "Thursday"
friday = "Friday"
saturday = "Saturday"

# An example of augmented assignment
spring, summer, winter, fall = "spring", "summer", "winter", "fall"

custom_end = "\n\n"

print(spring, summer, winter, fall)
print()

print(spring, summer, winter, fall, sep=" *^* ", end=custom_end)

print(sunday, monday, tuesday, wednesday, thursday, friday,
      saturday, sep="\n", end=custom_end)

print("The sum of 1", 2, 3, 4, 5, 6, 7, 8, 9, sep=" + ", end=" = 45\n")
```

```
$ python3 print_stuff.py
spring summer winter fall

spring *^* summer *^* winter *^* fall

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

The sum of 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45
$
```

The input() function in Python allows the programmer to prompt a user for input from the keyboard while the program is running.

The function accepts an option prompt as an argument that is displayed on the screen to the user.

- The prompt is displayed exactly as given.
  - ▶ No spaces or other formatting is added to the display.
- The function automatically removes the trailing newline from the string returned as the return value of the function.

getting_input.py

```
#!/usr/bin/env python3
prompt = "Please enter a number"
result = input(prompt)
print(result)
print()

# Slightly different approaches
print(prompt)
result = input(">")
print(result)
print()

print(prompt, "again")
print(input(">"))
```

```
$ python3 print_stuff.py
Please enter a number67
67

Please enter a number
>67
67

Please enter a number again
>67
67
$
```

Formatting Strings

Strings have a built in `%` operator that can be used to format stings similar to the sprintf() function in the C language.

- Use of the % operator with strings may lead to a number of common errors.
- An in depth description of the `%` operator for formatting strings can be found at the following URL:

```
https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting
```

Using the newer `format()` method and/or literal f-strings(PEP498) can help avoid these errors and provide a more powerful way to format text.

- The `format()` method is described in detail at the following URL:

```
https://docs.python.org/3/library/string.html#formatstrings
```

- F-strings are described in detail at the following URL:

```
https://docs.python.org/3/reference/lexical_analysis.html#f-strings
```

F-strings and the `format()` method share related syntax, though there are differences.

Format strings used by both f-strings and `format()` contain replacement fields surrounded by curly braces `{}`.

The general syntax is as follows (more complete syntax can be found in the links above).

- The replacement field is composed of an optional *field_name* followed by an optional `:` and *format_specification*.
  - The optional *field_name* is either a number or a keyword.
    - If it's a number, it refers to a positional argument (applies to `format()` but not f-strings)

    - If it's a keyword, it refers to a keyword argument when used with `format` or a existing variable when used in an f-string.

- The optional `:` and *format_specification* specify a non-default format for the replacement value.
    - The complete list of characters that make up the *format_specification* can be found at the following URL:

```
https://docs.python.org/3/reference/lexical_analysis.html#f-strings
```

The following example demonstrates various uses of both the `format()` method and f-strings to format strings.

string_formatting.py

```
#!/usr/bin/env python3

# Define format for both headers and data
fmt = "{:>20} : {}"

# Print column headers
print(fmt.format("America's Wars", "Dates"))
print(fmt.format("American Revolution", "1775 - 1783"))
print(fmt.format("War of 1812", "1812 - 1815"))
print(fmt.format("Mexican War", "1846 - 1848"))
print(fmt.format("Civil War", "1861 - 1865"))
print(fmt.format("Spanish American War", "1898"))
print(fmt.format("World War I", "1914 - 1918"))
print(fmt.format("World War II", "1939 - 1945"))
print(fmt.format("Korean War", "1939 - 1945"))
print(fmt.format("Vietnam War", "1939 - 1945"))
print(fmt.format("War on Terrorism", "2001 -"))
print(fmt.format("Iraq War", "2003 - 2011"))
print()
print("Source: roswellmemorialday.com")
print("roswellmemorialday.com/history/timline-of-americas-wars/")
print()

length = 10
width = 20
height = 5

print(f"A box with dimensions {length:02} x {width:02} x {height:02}",
end="")
print(f" has a volume of {length * width * height}")
```

```
$ python3 string_formatting.py
      America's Wars : Dates
 American Revolution : 1775 - 1783
         War of 1812 : 1812 - 1815
         Mexican War : 1846 - 1848
           Civil War : 1861 - 1865
Spanish American War : 1898
         World War I : 1914 - 1918
        World War II : 1939 - 1945
          Korean War : 1939 - 1945
         Vietnam War : 1939 - 1945
    War on Terrorism : 2001 -
            Iraq War : 2003 - 2011

Source: roswellmemorialday.com
roswellmemorialday.com/history/timline-of-americas-wars/

A box with dimensions 10 x 20 x 05 has a volume of 1000

$
```

Chapter 8. Intermediate Strings

Objectives

- Utilize strings methods to modify the case of strings.
- Identify the number of characters within a string.
- Find specific characters within a string.
- Modify the contents of a string.
- Use escape sequences.
- Create triple quoted strings.
- Work with string slices.

Strings

Strings in Python are an immutable sequence of zero or more characters.

- Literal string values may be enclosed in single or double quotes.
- Literal strings can span multiple lines in several ways.
  - This is achieved by using the line continuation character (`\`) as the last character.
- Literal strings may also be prefixed with a letter `r` or `R`.
  - These are referred to as raw strings and use different rules for backslash escape sequences.
- Strings in Python are represented using a class named `str`.
  - A class can be thought of as a new data type and an object as an instance of that data type.
- The `str` class has an abundance of methods defined within it.
  - Some of the methods return Boolean values such as `True` or `False`.
  - Some methods return a modified copy of the string.
  - A full listing of the string methods can be found in the online documentation or by using `help(str)` from the Python interactive shell.

String Casing Methods

Python provides several methods that can be used to modify the casing of characters in a string.

- These methods will return a copy of the string, the original string will not be changed.
- These methods can be used to modify the case of all characters in a string or the first character of each word in a string.
- `upper()` returns a copy of the string with all cased characters converted to uppercase.
- `lower()` returns a copy of the string with all cased characters converted to lowercase letters.
- `casefold()` returns a casefolded copy of the string.
  - It is similar to `lower()` but works on more characters and is intended to remove all case distinctions in a string.
  - It is often used for case insensitive comparisons.
- `title()` returns a copy of the string with the first letter of each word in the string in upper case and the rest of the characters in lower case.
- `capitalize()` returns a copy of the string with the first letter in upper case and the rest of the characters in lower case

string_case_methods.py

```
#!/usr/bin/env python3

oz = 'the WonderFul wiZard of OZ'
greeting = 'der Gruß'


print(f'{"original":10s}: {oz}')
print(f'{"upper":10s}: {oz.upper()}')
print(f'{"lower":10s}: {oz.lower()}')
print(f'{"title":10s}: {oz.title()}')
print(f'{"capitalize":10s}: {oz.capitalize()}')
print(f'{"original":10s}: {oz}')

print()

print(f'{"original":10s}: {greeting}')
print(f'{"casefold":10s}: {greeting.casefold()}')
print(f'{"lower":10s}: {greeting.lower()}')
print(f'{"original":10s}: {greeting}')
```

The output of the above program is shown below.

```
$ python3 string_case_methods.py
original  : the WonderFul wiZard of OZ
upper     : THE WONDERFUL WIZARD OF OZ
lower     : the wonderful wizard of oz
title     : The Wonderful Wizard Of Oz
capitalize: The wonderful wizard of oz
original  : the WonderFul wiZard of OZ

original  : der Gruß
casefold  : der gruss
lower     : der gruß
original  : der Gruß
$
```

Finding Sub-strings in a String

A variety of methods can be used to find instances of characters or strings within another string.

- `count(s)` will return the number of times the value passed to the method is found within the string.
- `index(s)` will return the index position of the first instance of the value passed to the method.
  - A `ValueError` exception is raised if the sub-string to find is not found within the string
- `find(s)` is similar to `index` but `find` will return a -1 when the sub-string is not found rather than raising an exception.
- Note that all these methods perform case sensitive searches.

string_finds.py

```
#!/usr/bin/env python3


data = input('Enter a sentence: ')

print(f'count("e"): {data.count("e")}')
print(f'index("e"): {data.index("e")}')
print(f'find("e"): {data.find("e")}')
print(f'find("E"): {data.find("E")}')
```

Sample output of the program is shown below.

```
$ python3 string_finds.py
Enter a sentence: the quick brown fox jumps over the lazy dog
count("e"): 3
index("e"): 2
find("e"): 2
find("E"): -1
$ python3 string_finds.py
Enter a sentence: Jill ran a marathon
count("e"): 0
Traceback (most recent call last):
  File "string_finds.py", line 7, in <module>
    print(f'index("e"): {data.index("e")}')
ValueError: substring not found
```

Modifying a String

Methods can be used to clean up data or to replace or remove unwanted characters. Each of these methods will return a copy of the string.

- `strip()` will remove any leading and trailing whitespace from the string.
  - `lstrip` will remove any leading whitespace (whitespace on the left)
  - `rstrip` will remove any trailing whitespace (whitespace on the right)
- `replace(old, new)` replaces all instances of the first parameter with all instances of the second parameter.
  - An optional third parameter can be used to specify the number of times the replacement is to be made.

string_cleaner.py

```
#!/usr/bin/env python3


phone_number = '\t  123-456-7890  \t'

print(f'{"original":10s}: {phone_number} : {len(phone_number)}')

cleaned = phone_number.strip()
print(f'{"strip":10s}: {cleaned} : {len(cleaned)}')

cleaned = phone_number.replace('-', '.')
print(f'{"replace":10s}: {cleaned} : {len(cleaned)}')

cleaned = phone_number.replace('-', '.', 1)
print(f'{"replace":10s}: {cleaned} : {len(cleaned)}')

cleaned = phone_number.replace('-', '')
print(f'{"replace":10s}: {cleaned} : {len(cleaned)}')

print(f'{"original":10s}: {phone_number} : {len(phone_number)}')
```

The output of the above program is shown below.

```
$ python3 string_cleaner.py
original  :       123-456-7890       : 18
strip     : 123-456-7890 : 12
replace   :       123.456.7890       : 18
replace   :       123.456-7890       : 18
replace   :       1234567890     : 16
original  :       123-456-7890       : 18
$
```

Escape Characters

The following is a list of escape sequences that can be used in literal strings to represent special characters.

Table 6. Escape Sequences

Sequence	Character/Meaning
`\newline`	Ignored
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	ASCII Bell (BEL)
`\b`	Backspace
`\f`	Form feed
`\n`	Linefeed
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\ooo`	ASCII character (octal value `ooo`)
`\xhhh`	ASCII character (hex value `hhh`)
`\uxxxx`	Unicode Character with 16-bit hex value `xxxx`
`\Uxxxxxxxx`	Unicode Character with 32-bit hex value `xxxxxxxx`

string_escapes.py

```
#!/usr/bin/env python3


print('\nThe instructor said \'JUMP!\' and we said \'How high?\'.\n')
print('\nThe instructor said \x27JUMP!\x27 and we said \047How
high?\047.\n')
print(r'\nThe instructor said \'JUMP!\' and we said \'How high?\'.\n')

print('\x28\x35\x2B\x39\x29\x2A\x33\x3D\x34\x32')

print()
print('\U0000265F', '\U0000265F', '\U0000265F', '\U0000265F',
      '\U0000265F', '\U0000265F', '\U0000265F', '\U0000265F')

print('\U0000265C', '\U0000265E', '\U0000265D', '\U0000265A',
      '\U0000265B', '\U0000265D', '\U0000265E', '\U0000265C')
```

Running the above program produces the following output:

```
$ python3 string_escapes.py

The instructor said 'JUMP!' and we said 'How high?'.


The instructor said 'JUMP!' and we said 'How high?'.

\nThe instructor said \'JUMP!\' and we said \'How high?\'.\n
(5+9)*3=42

♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
♜ ♞ ♝ ♚ ♛ ♝ ♞ ♜
$
```

Note that while the use of Unicode within Python 3 is fully supported, a font that is capable of displaying the characters is necessary to produce the above output.

Triple Quoted Strings

Python supports the use of triple quotes (single or double) for defining a long (or short) string.

```
s = '''this is a string'''

s = """this is also a string"""
```

- Triple quoted strings can also be used as `f-strings`.
- Any whitespace in the triple quoted string is included in the stored data and will be part of the printed output.
- Triple quoted strings are often used as docstrings, which are usually included as descriptions of functions and classes.
- Triple quoted strings are often used as a workaround for multi-line comments.
  - Note that triple quoted strings and comments are two different things.
    - Comments are ignored by the interpreter and are not stored in memory.
    - Triple quoted strings are *NOT* ignored by the interpreter and *ARE* stored in memory.

string_trips.py

```
#!/usr/bin/env python3


player_one = '''\U0000265F \U0000265F \U0000265F \U0000265F \U0000265F \
\U0000265F \U0000265F \U0000265F \n\U0000265C \U0000265E \U0000265D
\U0000265B \
\U0000265A \U0000265D \U0000265E \U0000265C'''

player_two = '''\U00002659 \U00002659 \U00002659 \U00002659 \U00002659 \
\U00002659 \U00002659 \U00002659 \n\U00002656 \U00002658 \U00002657 \
\U00002655 \U00002654 \U00002657 \U00002658 \U00002656'''

print(player_one)
print('''\n''' * 3)
print(player_two)
print("""\nYour move!""")
```

Running the above program produces the following output:

```
$ python3 string_escapes.py
♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜




♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖

Your move!
$
```

Indexing and Slicing

Strings, and other sequence types, have a set of operations that utilize a pair of square brackets (`[ ]`) in the syntax.

All three of the following types of operations return a sub-string of the sequence the operation is performed on.

- **Indexing**: Involves a single integer placed inside of the brackets.
  - Returns the character at the specified index position.
    - `the_string[index]`
- **Slicing**: Involves two integers separated by a colon in the brackets.
  - Returns a sub-string from the start index to the end index.
    - The end index is exclusive.
    - `the_string[start:stop]`
- **Extended Slicing**: Involves three integers separated by colons.
  - Returns a sub-string from the start index to the end index using the step.
    - The end index is exclusive.
    - `the_string[start:stop:step]`

The example below demonstrates the 3 types of operations listed above.

string_slicer.py

```
#!/usr/bin/env python3


state_capitol = 'Annapolis, Maryland'
pipe = ' | '

print(state_capitol)
# Indexing
print(state_capitol[0], state_capitol[4], state_capitol[-1], sep=pipe)

# Slicing
print(state_capitol[2:5], state_capitol[11:], state_capitol[:9], sep=pipe)

# Slicing from end
print(state_capitol[-4:-1], state_capitol[-4:], state_capitol[:-1],
sep=pipe)
print()

# Extended Slicing
alphabet = "abcdefghijklmnopqrstuvwxyz"
print(alphabet)
print(alphabet[2:22:5])
print(alphabet[::4])
```

Running the above program produces the following output:

```
$ python3 string_slicer.py
Annapolis, Maryland
A | p | d
nap | Maryland | Annapolis
lan | land | Annapolis, Marylan

abcdefghijklmnopqrstuvwxyz
chmr
aeimquy
$
```

Exercises

Exercise 1

Write a program that prompts a user to enter a string of text.

- The program should remove leading and trailing white space and then print:
  - The first character entered and the number of times it occurs in the string.
  - The last character entered and the number of times it occurs in the string.

Exercise 2

Write a program that prompts a user to enter a string of text.

- The program should remove all instances of the letter 'a'.
- The program should convert all lower case 'e' characters with upper case 'E' characters.
- The program should remove up to 2 lower case 'i' characters.
- The program should print the final string.

Exercise 3

Determine how many different string objects are stored in the variables below.

```
drink_a = "Iced Tea"
drink_b = "pop"
drink_c = 'water'
drink_d = "lemonade"
drink_e = "water'
drink_f = "iced tea"
drink_g = drink_c
drink_h = str(drink_b)
drink_i = drink_a + drink_b + drink_c + drink_d
drink_j = drink_a.lower()
drink_k = drink_f.capitalize()
drink_l = drink_f.title()
```

Exercise 4

Write a program that prompts a user to enter a whole number.

- Determine if the string contains all numeric characters.
  - An `if` statement is not required, string methods will suffice.

Exercise 5

Write a program that prompts a user to enter a floating point number.

- Determine if the string can be safely converted to a `float`
  - An `if` statement is not required, string methods will suffice.

Exercise 6

Write a program that prompts a user to enter a string of text and another prompt for a character or sequence of characters to search for in the text.

- The program should print:
  - The number of times the character or sequence of characters to search for occurs in the original string of text.
  - The first index position of the character or sequence of characters to search for.
    - The program should not crash if the character(s) to search for do not occur in the original text.

Exercise 7

Write a program that prompts a user to enter a string of text and another prompt for a character or sequence of characters to search for in the text.

- Make the program perform a case insensitive search.
- The program should print:
  - The number of times the character or sequence of characters to search for occurs in the original string of text.
  - The index position of the character or sequence of characters to search for.
    - The program should not crash if the character(s) to search for do not occur in the original text.

Exercise 8

Write a program that prompts a user to enter a url.

- The program should extract the host name and the domain into separate variables.
  - Print the host name and domain.

Exercise 9

Write a program that produces the following output.

```
Casey at the Bat by Ernest Thayer

    And somewhere men are laughing,
     and somewhere children shout,
    But there is no joy in Mudville
      mighty Casey has struck out!
```

Exercise 10

Write a program that prints a royal straight flush of spades.

Exercise 11

Write a program that requests a sentence, then print out the first and last words of the sentence, and finally, the number of words in the sentence.

- Any trailing punctuation, such as a period, should not be included in the last word.

Exercise 12

Write a program that takes a number as input, and prints out the ASCII character associated with that value.

Chapter 9. Lists

Objectives

- Use lists to store ordered collections of objects.
- Unpack list items.
- Use built-in functions on lists.
- Add and delete list items.
- Sort lists.
- Utilize join and split.
- Access individual list items and slices of lists.

Introduction to Lists

Python provides a `list` object, which is a sequence type that can hold a variety of objects.

- `list` objects can be described as
  - An ordered collection of zero or more elements.
    - Similar to an array in other languages.
  - Dynamic in that items can be added and removed.
- `list` objects support efficient element access using integer indices (slice notation).
- `list` objects can be created in several ways:
  - Using the `list()` constructor.
  - Using a pair of square brackets to denote the empty `list`
  - Using square brackets (`[]`), separating items with commas.
- Elements in a `list` can be a mix of any types.
- Elements in lists are accessed by their index positions.
  - Index positions are zero-based.
  - Elements can be retrieved and set by index.
    - `a_list[index]`
    - `a_list[index] = value`
- Slicing provides the ability to create a new `list` from a piece of an existing `list`.
  - Using sub-script notation, specify the start and end index.
  - The end index is exclusive.
    - `a_list[start:stop:step]`
  - All values for the start stop and step are optional.
- Available `list` methods can be found at

```
https://docs.python.org/3/tutorial/datastructures.html
```

creating_lists.py

```
#!/usr/bin/env python3


empty = list()
also_empty = []

numbers = [20, 3, 7, 82, -3, 45.6, 3, 65, 23]

pets = ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']

numbers_and_pets = numbers + pets

print(f'Empty lists: {empty} : {also_empty}')
print(f'Numbers: {numbers}')
print(f'Pets:  {pets}')
print(f'Numbers and pets: {numbers_and_pets}')

print(f'numbers[0] is {numbers[0]}')
print(f'pets[2] is {pets[2]}')

slice = numbers_and_pets[7:11]
print(f'Slice: {slice}')

print(f'Some pets: {pets[1:4]}')
print(f'Last pets: {pets[-3:-1]}')
```

The output of the above program is shown next.

```
$ python3 creating_lists.py
Empty lists: [] : []
Numbers: [20, 3, 7, 82, -3, 45.6, 3, 65, 23]
Pets:  ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']
Numbers and pets: [20, 3, 7, 82, -3, 45.6, 3, 65, 23, 'Moose', 'Jynx',
'Morrigan', 'Sadie', 'Duke', 'Gizmo']
numbers[0] is 20
pets[2] is Morrigan
Slice: [65, 23, 'Moose', 'Jynx']
Some pets: ['Jynx', 'Morrigan', 'Sadie']
Last pets: ['Sadie', 'Duke']
$
```

Introduction to Lists

Attempting to access elements beyond the bounds of the `list` will result in an `IndexError` being raised.

- The bounds of the `list` is the index position of the last element.
- The built-in `len` function will return the number of elements in a `list` when a `list` is passed to it.
  - `len(a_list)`
  - The last index position will always be one less than the length of the `list`
    - Alternatively, use negative indexing to get the last element in the `list`

Unpacking Lists

The elements of a list can be unpacked into individual variables that are often easier to understand rather than referencing them by index.

- When unpacking lists, the number of variables being assigned values to must be the same as the number of elements in the list.
- The variables are copies of the data in the list.
  - Changes made to the variables do not affect the list.
  - Changes made to the list after unpacking do not affect the variables.

unpacking_lists.py

```
#!/usr/bin/env python3


weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
mon, tue, wed, thu, fri = weekdays

print(weekdays)
print(mon, wed, fri)

wed = 'Hump day!'

print(wed)
print(weekdays)

weekdays[2] = 'grey'
print(wed)
print(weekdays)
```

```
$ python3 unpacking_lists.py
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
Monday Wednesday Friday
Hump day!
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
Hump day!
['Monday', 'Tuesday', 'grey', 'Thursday', 'Friday']
$
```

Dynamic Lists

A `list` is dynamic and as such can grow and/or shrink as needed.

- Adding, deleting, and updating elements are common operations performed on a `list`.
- `list` objects provide several methods that can be used to add or update elements within the `list`.
  - `append(item)`
    - Adds a new element to the end of the `list`, increasing the length by one.
  - `extend(iterable)`
    - Adds each element from the iterable passed in as new elements to the end of the `list`, increasing the length by the number of items in the iterable.
  - `insert(index, item)`
    - Adds the specified item at the specified index position, shifting existing elements to make room for the new item, increasing the length by one.
  - `pop([index])`
    - Deletes the element at the specified index or the end of the `list` if no index is provided, decreasing the length by one.
  - `remove(value)`
    - Deletes the first instance of the specified value in the `list`, decreasing the length by one.

dynamic_lists.py

```
#!/usr/bin/env python3


pets = ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']
more_pets = ['Tinkerbell', 'Mittens', 'Snarf', 'Jynx']

print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

# adding elements
pets.append('Bizbark')
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

pets.extend(more_pets)
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

pets.insert(3, 'Kaiser')
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

# deleting elements
pets.pop()
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

pets.pop(5)
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')

pets.remove('Jynx')
print(f'Pets length: {len(pets)}. Pets:\n{pets}\n{"-" * 22}')
```

The output of the above program is shown next.

```
$ python3 dynamic_lists.py
Pets length: 6. Pets:
['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']
----------------------
Pets length: 7. Pets:
['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo', 'Bizbark']
----------------------
Pets length: 11. Pets:
['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo', 'Bizbark',
'Tinkerbell', 'Mittens', 'Snarf', 'Jynx']
----------------------
Pets length: 12. Pets:
['Moose', 'Jynx', 'Morrigan', 'Kaiser', 'Sadie', 'Duke', 'Gizmo',
'Bizbark', 'Tinkerbell', 'Mittens', 'Snarf', 'Jynx']
----------------------
Pets length: 11. Pets:
['Moose', 'Jynx', 'Morrigan', 'Kaiser', 'Sadie', 'Duke', 'Gizmo',
'Bizbark', 'Tinkerbell', 'Mittens', 'Snarf']
----------------------
Pets length: 10. Pets:
['Moose', 'Jynx', 'Morrigan', 'Kaiser', 'Sadie', 'Gizmo', 'Bizbark',
'Tinkerbell', 'Mittens', 'Snarf']
----------------------
Pets length: 9. Pets:
['Moose', 'Morrigan', 'Kaiser', 'Sadie', 'Gizmo', 'Bizbark', 'Tinkerbell',
'Mittens', 'Snarf']
----------------------
$
```

Sorting Collections

Basic sorting of a `list` is provided by the `sort()` method that sorts a list in place, and the built-in `sorted()` function that takes an iterable object as its parameter and returns a new sorted `list`.

- There are a few ways to use them to sort data in various types of collections.
- A keyword parameter, `reverse=True`, can be passed to reverse the natural order of the objects (typically being ascending order).
- A customized sort can be achieved by passing another keyword parameter named `key` to specify a function to be used for sorting.
  - The function must have one parameter and return a value.
    - The return value is used for the sorting order.
- There are some fundamental differences between the `list.sort()` method and the `sorted(iterable)` function.
  - The `list.sort()` method is an instance method and belongs to a specific `list` object.
  - `list.sort()` will modify the `list` object to which it belongs.
    - It does not return a new `list`.
  - The `sorted` function does not belong to a specific `list` object.
  - The `sorted` function can sort any iterable.
  - The `sorted` function returns a `list` and does not modify the iterable that was passed in as an argument.

The example that follows demonstrates basic sorting of a list in both ascending and descending order.

basic_sorting.py

```
#!/usr/bin/env python3


numbers = [3, 1, -10, 54, 75, 29, 4.2]
pets = ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']

print(f'Numbers: {numbers}')
numbers.sort()
print(f'Numbers.sort: {numbers}')
numbers.sort(reverse=True)
print(f'Reverse numbers.sort: {numbers}')

print()

print(f'Pets: {pets}')
pets.sort()
print(f'Pets.sort: {pets}')
pets.sort(reverse=True)
print(f'Reverse pets.sort: {pets}')

print()

evens = [4, 10, 8, 2, 6]
print(f'Evens: {evens}')
sorted_evens = sorted(evens)
print(f'sorted_evens: {sorted_evens}')
print(f'Evens: {evens}')

print()

greek = ['gamma', 'beta', 'zed', 'delta', 'alpha']
print(f'greek: {greek}')
sorted_greek = sorted(greek, reverse=True)
print(f'Reverse sorted_greek: {sorted_greek}')
print(f'greek: {greek}')
```

```
$ python3 basic_sorting.py
Numbers: [3, 1, -10, 54, 75, 29, 4.2]
Numbers.sort: [-10, 1, 3, 4.2, 29, 54, 75]
Reverse numbers.sort: [75, 54, 29, 4.2, 3, 1, -10]

Pets: ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']
Pets.sort: ['Duke', 'Gizmo', 'Jynx', 'Moose', 'Morrigan', 'Sadie']
Reverse pets.sort: ['Sadie', 'Morrigan', 'Moose', 'Jynx', 'Gizmo', 'Duke']

Evens: [4, 10, 8, 2, 6]
sorted_evens: [2, 4, 6, 8, 10]
Evens: [4, 10, 8, 2, 6]

greek: ['gamma', 'beta', 'zed', 'delta', 'alpha']
Reverse sorted_greek: ['zed', 'gamma', 'delta', 'beta', 'alpha']
greek: ['gamma', 'beta', 'zed', 'delta', 'alpha']
$
```

- Note that calling `sort()` on a `list` updates the `list` in place.
- If the desire is to obtain a sorted version of the list and leave the original list unchanged, then the built-in `sorted()` function can be used instead.
- Sorting lists will raise an exception if the list contains data types that cannot be compared to each other.
  - For example, sorting will fail if the `list` contains integers and strings.

Custom Sorting

As mentioned earlier, a customized sort can be achieved by passing a named argument called `key`.

- The value of the `key` keyword parameter should be a reference to a function that takes a single argument.
- The return value of the function, when invoked by the `sort()` method or `sorted()` function, will then be used as the comparison key from each element in the list being sorted.
  - The `sorted()` function utilizes the same `reverse` and `key` optional named arguments as the `list.sort)` method.

The next example demonstrates sorting a list of strings by the length of each string.

- This is accomplished by passing the built-in `len()` function as the keyword parameter value to the list's sort method `sort(key=len)`

custom_sorting.py

```
#!/usr/bin/env python3


names = '''Smith Johnson Williams Brown Jones Miller Lee
Garcia Rodriguez Wilson Martinez Anderson Taylor
Thomas Hernandez Moore Martin Jackson Thompson
White Lopez Davis'''

names = names.split()

# Primary sort by name
names.sort()
print(names)

print()

# Secondary sort by length
names.sort(key=len)
print(names)
```

The output of the above program is shown next.

```
$ python3 custom_sorting.py
['Anderson', 'Brown', 'Davis', 'Garcia', 'Hernandez', 'Jackson',
'Johnson', 'Jones', 'Lee', 'Lopez', 'Martin', 'Martinez', 'Miller',
 'Moore', 'Rodriguez', 'Smith', 'Taylor', 'Thomas', 'Thompson',
 'White', 'Williams', 'Wilson']

['Lee', 'Brown', 'Davis', 'Jones', 'Lopez', 'Moore', 'Smith','White',
'Garcia', 'Martin', 'Miller', 'Taylor', 'Thomas', 'Wilson', 'Jackson',
'Johnson', 'Anderson', 'Martinez', 'Thompson', 'Williams','Hernandez',
'Rodriguez']
$
```

The two sorts together in the application above results in any names of the same length being sorted alphabetically.

Joining and Splitting

`str` objects have methods that can be used to create a `list` from a `str` and concatenate `str` objects in a `list` to a string.

- The `join` method concatenates `str` objects from an iterable object and returns them as a single `str`.
  - `str.join(values)`
  - The separator in the returned `str` is the `str` to which this method belongs.
  - A `TypeError` exception is raised if the iterable object contains non-string values.
- The `split` method returns a `list` of the words in a `str` object.
  - `str.split(sep=Non, maxsplit=-1)`
  - If no value is specified for `sep`, then whitespace is used as the separator
    - the resulting `list` will not contain any empty strings.
  - If a value is specified for `sep`, consecutive delimiters are not grouped which could result in empty strings in the returned `list`.
  - `maxsplit` is used to determine the number of times the split should occur.
    - If no value is supplied, all possible splits are made.

join_splits.py

```
#!/usr/bin/env python3


skit = 'I say, Who\'s on first, What\'s on second, I Don\'t Know\'s on
third'
print(skit)
skit_list = skit.split(', ')
print(type(skit_list), skit_list)

data = 'this    is some      data'
print(data)
print(data.split(' '))
print(data.split())

pets = ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo', 7]
print(f'Pets:  {pets}')

joined_pets = ' - '.join(pets)
print(joined_pets)
```

The output of the above program is shown next.

```
$ python3 join_splits.py
I say, Who's on first, What's on second, I Don't Know's on third
<class 'list'> ['I say', "Who's on first", "What's on second", "I Don't
Know's on third"]
this    is some      data
['this', '', '', '', 'is', 'some', '', '', '', '', '', 'data']
['this', 'is', 'some', 'data']
Pets:  ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']
Moose - Jynx - Morrigan - Sadie - Duke - Gizmo
$
```

min, max, and sum

A common problem to solve is to find the largest or smallest value in a list. As is the ability to calculate the total of all values in a list. Python provides built-in functions to solve these problems.

- max(iterable, [key], [default])
  - In it's simplest form, max returns the largest item in an iterable.
  - key is an optional parameter.
  - default is also an optional parameter.
    - If the iterable is empty and a default is *NOT* provided, a ValueError is raised.
    - If the iterable is empty and a default *IS* provided, the default value is returned.
- min(iterable, [key], [default])
  - min behaves the same as max except it returns the smallest item in an iterable.
  - It has the same optional arguments with the same behavior.
- sum(iterable, start=0)
  - Sums the value of start and each item in an iterable.
  - start is an optional parameter.
  - The items in the iterable must be numeric.
- All three of these functions will raise an TypeError if the iterable contains numbers and strings.
- max and min will return the string with the largest or smallest ASCII value at the zero index position.

math_func.py

```
#!/usr/bin/env python3


empty = []
numbers = [3, 1, -10, 54, 75, 29, 4.2]
pets = ['Moose', 'Jynx', 'Morrigan', 'Sadie', 'Duke', 'Gizmo']

print(f'Empty: {empty}')
print(f'Max empty is {max(empty, default=None)}')
print(f'Min empty is {min(empty, default=None)}')
print(f'Sum of empty is {sum(empty, start=100)}')

print()

print(f'Numbers: {numbers}')
print(f'Max number is {max(numbers)}')
print(f'Min number is {min(numbers)}')
print(f'Sum of numbers is {sum(numbers)}')

print()

animals = ['bee', 'zebu', 'albatross', 'cat', 'zebra']
print(f'animals: {animals}')
print(f'Max animals is {max(animals)}')
print(f'Min animals is {min(animals)}')

print(f'Max animals key len is {max(animals, key=len)}')
print(f'Min animals key len is {min(animals, key=len)}')
```

The output of the above program is shown next.

```
$ python3 math_func.py
Empty: []
Max empty is None
Min empty is None
Sum of empty is 100

Numbers: [3, 1, -10, 54, 75, 29, 4.2]
Max number is 75
Min number is -10
Sum of numbers is 156.2

animals: ['bee', 'zebu', 'albatross', 'cat', 'zebra']
Max animals is zebu
Min animals is albatross
Max animals key len is albatross
Min animals key len is bee
$
```

all and any

Another common problem to solve is determining if all elements in a list are True or if any element in the list is True. Python provides built-in functions for this purpose.

- all(iterable)
  - The all function returns True if every element in the iterable is True or if the iterable is empty.
- any(iterable)
  - The any function returns True if any element in the iterable is True.
  - False will be returned if the iterable is empty.

all_any.py

```
#!/usr/bin/env python3


empty = []
numbers = [0, 1, 2, 3]
odds = [1, 3, 5, 7, 9]
words = ['this', '']

trues = [True, True, True]
falses = [False, False, False]
both = trues + falses

print(f'Empty: {empty}')
print(f'All empty is {all(empty)}')
print(f'Any empty is {any(empty)}')

print()

print(f'Numbers: {numbers}')
print(f'All numbers is {all(numbers)}')
print(f'Any numbers is {any(numbers)}')

print()
```

```
print(f'Words: {words}')
print(f'All words is {all(words)}')
print(f'Any words is {any(words)}')

print()

print(f'Trues: {trues}')
print(f'All trues is {all(trues)}')
print(f'Any trues is {any(trues)}')

print()

print(f'Falses: {falses}')
print(f'All falses is {all(falses)}')
print(f'Any falses is {any(falses)}')

print()

print(f'Both: {both}')
print(f'All both is {all(both)}')
print(f'Any both is {any(both)}')
```

The output of the above program is shown next.

```
$ python3 all_any.py
Empty: []
All empty is True
Any empty is False

Numbers: [0, 1, 2, 3]
All numbers is False
Any numbers is True

Words: ['this', '']
All words is False
Any words is True

Trues: [True, True, True]
All trues is True
Any trues is True

Falses: [False, False, False]
All falses is False
Any falses is False

Both: [True, True, True, False, False, False]
All both is False
Any both is True
$
```

Lists Containing Lists

Items within a `list` can be any Python object. This includes other collections, custom classes, etc.

- When working with a `list` as an item in a `list`, items can be accessed using indexing
- If only one index is supplied then the inner `list` will be returned
- Two sets of square brackets are required to access an item in the inner `list`.

list_in_list.py

```
#!/usr/bin/env python3


multi = [[100, 101, 103], [200, 201, 202], [300, 301, 302]]

print(multi)
print(f'multi[0]: {multi[0]}')
print(f'multi[0][0]: {multi[0][0]}')
print(f'multi[1][1]: {multi[1][1]}')
print(f'multi[-1][-1]: {multi[-1][-1]}')
```

The output of the above program is shown next.

```
$ python3 list_in_list.py
[[100, 101, 103], [200, 201, 202], [300, 301, 302]]
multi[0]: [100, 101, 103]
multi[0][0]: 100
multi[1][1]: 201
multi[-1][-1]: 302
$
```

Exercises

Exercise 1

Prompt the user to enter several words separated by semicolons.

- Create a `list` from the user's input.
  - Each word should be an element in the `list`.
- The program should print the entire `list` sorted alphabetically.
- The program should also print the number of words entered.

Exercise 2

Write a program to get a `list` of words and print them in descending alphabetical order.

- The program should then print the `list` with words sorted by the length of each word with the longest word being first.

Exercise 3

Re-write the previous exercise but instead of modifying the `list`, the program should sort a copy of the `list`.

- The program should print the original `list` and the sorted `list`.

Exercise 4

Create a list of numbers.

- The program should print `True` if there is not a zero in the `list`.
- The program should create a `list` that contains the following information about the `list` of numbers:
  - The total of all the numbers in the `list`.
  - The average of the numbers in the `list`.
  - The largest number in the `list`.
  - The smallest number in the `list`.

Exercise 5

Prompt the user to enter several words separated by commas.

- Create a `list` from the user's input.
  - Each word should be an element in the list.
- The program should print `False` if there is an empty string in the `list` and `True` if there is not an empty string.
- The program should print the list in reverse order, without using the `list.reverse()` method.

Exercise 6

Given this list,

```
caps = [['Arizona', 'Phoenix', 1445632], ['Indiana', 'Indianapolis', 829718],
['Ohio', 'Columbus', 822553]]
```

- Create a program that generates the following output: `Phoenix:1445632 - Indianapolis:829718`

Exercise 7

Get a `list` of words from a user.

- The program should print the longest word in the `list`, along with the index position and the

length of the longest word in the `list`.

- The program should print the second shortest word in the list, along with the index position and length of the second shortest word in the list.
  - The index position should be where it was entered by the user.

Chapter 10. Dictionaries

Objectives

- Use dictionaries to store key/value pairs.
- Add and delete key/value pairs.
- Retrieve individual values.
- Retrieve all keys, values, and key/value pairs.
- Reverse and sort dictionaries.

Introduction to Dictionaries

A `dict` (dictionary) is an ordered collection of entries.

- Each entry contains a key/value pair.
- Dictionaries maintain their order of insertion.
  - Last In, First Out (LIFO)
  - Prior to Python 3.6, dictionaries were unordered.
- Dictionary keys have to be both unique and hashable.
  - All of Python's immutable built-in objects are hashable and can be used as keys in a dictionary.
- Dictionaries can be created in several ways.
  - By placing a comma-separated list of key-value pairs within braces.

```
tickers = {'UA': 'Under Armour', 'AMZN': 'Amazon.com, Inc.', 'SAP': 'SAP
AG'}
positions = {1: "First", 2: "Second", 3: "Third"}
empty = {}
```

- By calling the `dict()` constructor.

```
empty_also = dict()
something = dict(key1='value 1', key2='value 2')
```

- All available dictionary methods can be found at

```
https://docs.python.org/3/library/stdtypes.html#typesmapping
```

Working With Individual Key/Value Pairs

Dictionaries are designed as a data structure that provides fast lookups into the structure by key. Sub-script notation can be used to add new key/value pairs and to modify existing values for a given key.

- `dictionary[key]` will retrieve the value associated to the given key.
  - If the key does not exist, a `KeyError` exception is raised
- `dictionary[key = value]` will either:
  - Add a new key value pair.
  - Or modify the value associated to the given key.
- Dictionaries have a `get` method that can be used to retrieve the value for a given key.
  - `dictionary.get(key[, default])`
    - The optional `default` parameter defines the return value of the method if the given key does not exist in the dictionary.
    - If a value for `default` is not provided, `None` will be returned.

dictionary_basics.py

```
#!/usr/bin/env python3


tickers = {'UA': 'Under Armour', 'SAP': 'SAP AG', 'AMZN': 'Amazon'}

# get specific values from the dictionary
print('tickers[SAP]:', tickers['SAP'])
print('tickers.get(AMZN):', tickers.get('AMZN'))
print('tickers.get(MSFT):', tickers.get('MSFT'))
print('tickers.get(MSFT, Key not found):', tickers.get('MSFT', 'Key not
found'))
print('-' * 70)

# add new key/value pairs
tickers['FIT'] = 'Fitbit, Inc.'
tickers['APPL'] = 'Apple, Inc.'
print(tickers)
print('-' * 70)

# modify an existing key/value pair
tickers['AMZN'] = 'Amazon.com, Inc.'
print(tickers)
print('-' * 70)
```

The output of the above program is shown next.

```
$ python3 dictionary_basics.py
tickers[SAP]: SAP AG
tickers.get(AMZN): Amazon
tickers.get(MSFT): None
tickers.get(MSFT, Key not found): Key not found
------------------------------------------------------------------------
{'UA': 'Under Armour', 'SAP': 'SAP AG', 'AMZN': 'Amazon', 'FIT': 'Fitbit,
Inc.', 'APPL': 'Apple, Inc.'}
------------------------------------------------------------------------
{'UA': 'Under Armour', 'SAP': 'SAP AG', 'AMZN': 'Amazon.com, Inc.', 'FIT':
'Fitbit, Inc.', 'APPL': 'Apple, Inc.'}
------------------------------------------------------------------------
$
```

Removing Key/Value Pairs

Removing elements from a dictionary can be done through the `pop()` and `popitem()` methods.

- The general syntax for the `pop()` method is as follows.
  `value = dictionary.pop(key[,default])`
  - `key` represents the key of the key/value pair to attempt to remove.
  - `default` represents an optional value to return if key does not exist.
    - A `KeyError` exception is raised if no default is given and the `key` does not exist.
- The general syntax for the `popitem()` method is as follows.
  `a_tuple = dictionary.popitem()`
  - The `popitem()` method removes and returns the last key/value pair as a `tuple`, but raises a `KeyError` if `obj` is empty.
    - A `tuple` is an immutable collection of ordered elements.
- The `del` command can also be used to remove a key/value pair from the dictionary or the entire dictionary.
  - `del dictionary[key]` removes the specified key/value pair from the dictionary.
    - The key/value pair is not returned.
    - A `KeyError` is raised if the specified key is not in the dictionary.
  - `del dictionary` deletes the entire dictionary.

removing_from_dictionary.py

```
#!/usr/bin/env python3


tickers = {'UA': 'Under Armour', 'SAP': 'SAP AG', 'AMZN': 'Amazon',
           'FIT': 'Fitbit, Inc.', 'APPL': 'Apple, Inc.'}

print(f'tickers: {tickers}')
# pop
removed = tickers.pop('UA')
print(f'pop(UA): {removed}')
print(f'tickers: {tickers}')
print('-' * 70)

# popitem
removed = tickers.popitem()
print(f'popitem(): {removed}')
removed = tickers.popitem()
print(f'popitem(): {removed}')
print(f'tickers: {tickers}')
print('-' * 70)

# pop something that doesn't exist with default return value
removed = tickers.pop('ABC', 'Key does not exist')
print(f'pop(ABC, Key does not exist): {removed}')
print('-' * 70)

# pop something that doesn't exist
removed = tickers.pop('ABC')
print(f'pop(ABC): {removed}')

print(f'ending tickers: {tickers}')
```

The output of the above program is shown next.

```
$ python3 removing_from_dictionary.py
tickers: {'UA': 'Under Armour', 'SAP': 'SAP AG', 'AMZN': 'Amazon', 'FIT':
'Fitbit, Inc.', 'APPL': 'Apple, Inc.'}
pop(UA): Under Armour
tickers: {'SAP': 'SAP AG', 'AMZN': 'Amazon', 'FIT': 'Fitbit, Inc.',
'APPL': 'Apple, Inc.'}
------------------------------------------------------------------------
popitem(): ('APPL', 'Apple, Inc.')
popitem(): ('FIT', 'Fitbit, Inc.')
tickers: {'SAP': 'SAP AG', 'AMZN': 'Amazon'}
------------------------------------------------------------------------
pop(ABC, Key does not exist): Key does not exist
------------------------------------------------------------------------
Traceback (most recent call last):
  File "removing_from_dictionary.py", line 28, in <module>
    removed = tickers.pop('ABC')
KeyError: 'ABC'$
```

Working With All Key/Value Pairs

When there is a need to work with all key/value pairs in a dictionary, the following methods are useful.

- The `keys()` method returns a view of the dictionary keys.
- The `values()` method returns a view of the dictionary values.
- The `items()` method returns a view of the dictionary items.
  - Each item in the view returned by `items()` is a 2-`tuple` consisting of a key and a value.
- The objects returned by the methods are referred to as view objects.
  - A view object provides a dynamic view of the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.
  - The views returned by the above methods are iterable, as is the dictionary.
  - Each of the views can also be converted to a list or a tuple by passing the view to a `list()` or `tuple()` constructor.
  - The views returned by `keys()` and `items()` are set-like views that can be converted to sets with the `set()` constructor.

dictionary_views.py

```
#!/usr/bin/env python3


lotteries = {'Mega Millions': 227000000, 'Power Ball': 40000000,
             'Multi Match': 625000}

# get all keys
the_keys = lotteries.keys()
print(f'the_keys: {the_keys}')
print('-' * 50)

# get all values
the_values = lotteries.values()
print(f'the_values: {the_values}')
print('-' * 50)


# get all key/value pairs
the_items = lotteries.items()
print(f'the_items: {the_items}', )
print('-' * 50)

lotteries['Bonus Match 5'] = 50000
lotteries['Mega Millions'] = 141000000
lotteries.pop('Multi Match')

print(f'the_keys: {the_keys}')
print(f'the_values: {the_values}')
print(f'the_items: {the_items}', )
```

The output of the above program is shown next.

```
$ python3 dictionary_views.py
the_keys: dict_keys(['Mega Millions', 'Power Ball', 'Multi Match'])
-------------------------------------------------
the_values: dict_values([227000000, 40000000, 625000])
-------------------------------------------------
the_items: dict_items([('Mega Millions', 227000000), ('Power Ball',
40000000), ('Multi Match', 625000)])
-------------------------------------------------
the_keys: dict_keys(['Mega Millions', 'Power Ball', 'Bonus Match 5'])
the_values: dict_values([141000000, 40000000, 50000])
the_items: dict_items([('Mega Millions', 141000000), ('Power Ball',
40000000), ('Bonus Match 5', 50000)])
$
```

Finding Keys and Values

The `in` and `not in` operators can be used to determine if a specific key or value exists in the dictionary.

- Both operators will return a `True` or `False` value.
- When used with the entire dictionary, the operators search for the value in the keys of the dictionary.
- The view returned by the `values` method can be used to search for a value in the values of the dictionary.

dictionary_finds.py

```
#!/usr/bin/env python3


lotteries = {'Mega Millions': 227000000, 'Power Ball': 40000000,
             'Multi Match': 625000}

print('Mega Millions' in lotteries)
print('Mega Millions' not in lotteries)
print('Bonus Match 5' in lotteries)
print('Bonus Match 5' not in lotteries)
print(625000 in lotteries)
print(625000 in lotteries.values())
print(1 in lotteries.values())
```

The output of the above program is shown next.

```
$ python3 dictionary_finds.py
True
False
False
True
False
True
False
$
```

Sorting Dictionaries

Dictionaries cannot be sorted; however, the keys and values can be passed to the `sorted` function. A sorted `list` will be returned.

- The keys and values of a dictionary can be converted to a `list`, which could then be sorted by using the `sort` method of the `list`.
- Alternatively, the `sorted` function can be used.
- To sort the resulting `list` objects by the values in the dictionary, use the `get` method of the dictionary as the `key` of the `sort` method and the `sorted` function.

dictionary_sorts.py

```
#!/usr/bin/env python3
def get_value(akey):
    return states[akey]


states = {'Massachusetts': 'MA', 'New Hampshire': 'NH', 'Maryland': 'MD',
          'Nevada': 'NV', 'Maine': 'ME'}

sorted_states = list(states.keys())
sorted_states.sort()
print(f'States by long name: {sorted_states}')
sorted_states = list(states.keys())
sorted_states.sort(key=states.get)
print(f'States by short name: {sorted_states}')

sorted_states = sorted(states.keys())
print(f'States by long name: {sorted_states}')
sorted_states = sorted(states.keys(), key=states.get)
print(f'States by short name: {sorted_states}')

sorted_states = sorted(states.keys(), key=states.get, reverse=True)
print(f'States by short name reversed: {sorted_states}')
```

The output of the above program is shown next.

```
$ python3 dictionary_sorts.py
States by long name: ['Maine', 'Maryland', 'Massachusetts', 'Nevada', 'New
Hampshire']
States by short name: ['Massachusetts', 'Maryland', 'Maine', 'New
Hampshire', 'Nevada']
States by long name: ['Maine', 'Maryland', 'Massachusetts', 'Nevada', 'New
Hampshire']
States by short name: ['Massachusetts', 'Maryland', 'Maine', 'New
Hampshire', 'Nevada']
States by short name reversed: ['Nevada', 'New Hampshire', 'Maine',
'Maryland', 'Massachusetts']
$
```

Exercises

Exercise 1

Create a dictionary that has the following characters as keys and their hex equivalent as the value:

```
! @ # $ % ^ & * ( )
```

- Print the keys of the dictionary.
- Prompt the user to enter a single special character to add to the dictionary.
  - Add the character and it's hex value to the dictionary.
- Prompt the user for a character to retrieve from the dictionary.
  - The program should print value associated to the character the user inputs.
  - The program should not crash if they provide a key that does not exist in the dictionary.

Exercise 2

Create a dictionary that has several key/value pairs of states as the key and the capitol as the value.

- Prompt the user for a state and capitol pair to add to the dictionary.
- Print the states sorted alphabetically by the keys.
- Prompt the user for a state and print the capitol.
  - The program should not raise an exception if the state is not found.
- Print the capitols sorted by the longest name to the shortest name.

Exercise 3

Use a dictionary to create a mapping of the digits 0-9 to the words 'zero', 'one,' 'two,' etc.

- Prompt the user to enter a number.
- The program should identify if the value the user entered is contained within the dictionary or not.
- The program should print the word of the digit the user entered.
  - The program should not raise an exception if the digit is not in the dictionary.

Exercise 4

Create a dictionary that has several key/value pairs of states as the key and a tuple that contains the capitol and population as the value.

- Prompt the user for a state, capitol and population to add to the dictionary.
- Print the states sorted alphabetically by the keys.
- Prompt the user for a state and print the capitol and population. The population should have commas where appropriate.
    - The program can raise an exception if the state is not found or the population is not a numeric value.

Exercise 5

Create a program that prompts a user for the following information and stores it in a dictionary with the same string as the key:

listing

```
chore
verb
famous_person
job
location
nickname
```

- Using the values from the dictionary, create a mad lib for the following string and print the resulting song.
    - Song credit goes to 'Weird Al' Yankovic, It’s All About the Pentiums

```
song = f'''What y'all wanna do?
Wanna be {job}? Code crackers? Slackers
Wastin' time with all the {location} yakkers
9 to 5, {verb} at Hewlett Packard
...
Yeah, payin' the bills with my mad {job} skills
...
I'm down with {famous_person}, I call him "{nickname}" for short
I phone him up at home and I make him do my {chore}'''
```

Exercise 6

Create a program that has a dictionary that has the following words as keys and their appropriate pronunciations as the values.

```
valgrind is pronounced 'val-grinned'
gif is pronounced 'jiff'
SQL is pronounced 'es queue el'
JSON is pronounced 'jayson' (although Douglas Crockford doesn't really
care_)
```

- The program should prompt a user for a word and display the appropriate pronunciation.
  - ▶ The program should not raise an exception if the user enters a word that is not in the dictionary.
- Prompt the user for a word and delete the key/value pair from the dictionary.
  - ▶ The program should not raise an exception if they attempt to delete a key/value pair that does not exist.
- Delete the last key/value pair added to the dictionary.
- Print the dictionary.

Chapter 11. Conditionals

Objectives

- Use the indenting requirements of Python properly.
- Use the Python control flow constructs correctly.
- Understand and use Python’s various relational and logical operators.
- Use if statements to make decisions within the Python code.

Indenting Requirements

Python provides a robust set of keywords and related items that control the flow of execution within an application.

- In this section, we will explore the various conditional execution options that Python provides.
- In addition, we will look at the various operators used by Python in control flow constructs.
- **Python mandates the use of indenting within a compound statement.**
  - The first line of the compound statement is referred to as the header
  - All other statements within the compound statement are referred to as the suite or body and must be indented the same number of columns to be part of the same suite.
  - The suite ends with the first statement that is "dedented" to the column of the header.
- One such type of compound statement is a control structure.
- Here is the general syntax for an `if` statement.

```
if some_condition:
    suite_statement_1
    suite_statement_2
    suite_statement_3 # suite ends here

print("some output")
```

- Suites must be indented the same amount of white space from the starting column of the header.
- If tabs are used in the source code, a single tab is not equal to the number of spaces used in a tab.
- It is recommended that spaces be used over tabs.
- Typically, many editors and IDEs will automatically indent for you.

The if Statement

The fundamental decision making control structure in many programming languages is the `if` statement.

- Code inside an `if` suite is only executed if a given condition is `True`
- The following examples demonstrate proper indenting when using the `if` statement and its variants.
- Also, notice the required use of the colon (`:`) to end the header portion of the `if` and the `else`.

if_else.py

```
#!/usr/bin/env python3


number = int(input('Enter a number between 1 and 100: '))
target = 50
if number < target:
    print(f'{number} is less than {target}')
else:
    print(f'{number} is greater than or equal to {target}')
```

In the above example, `number` and `target` are being printed in both the `if` and the `else` statement.

The output of the above program is shown next.

```
$ python3 if_else.py
Enter a number between 1 and 100: 42
42 is less than 50
$
```

elif and else

When writing an if statement, there can be zero or more elif blocks, and the else block is optional.

- The keyword elif can be used to prevent the testing of multiple if statements when only one of the if statements can ever be True at a time.
  - It can also sometimes prevent the need for nesting if statements inside of an else.
- Since Python does not have a "switch" statement found in other languages, elif is often a suitable substitute.
  - elif gives the opportunity to have multiple conditional checks.
  - When one conditional check results in True, the True part of the suite is executed and the rest of the elif statements are skipped.

elif_example.py

```
#!/usr/bin/env python3
value = int(input('Please enter a whole number: '))

print(value, end=' is ')
if value <= 5:
    print('less than or equal to 5')
elif value <= 10:
    print('between 6 and 10 inclusively')
elif value <= 15:
    print('between 11 and 15 inclusively')
else:
    print('greater than 15')

print('This statement is not part of the above if else')
```

The output of the above program is shown next.

```
$ python3 elif_example.py
Please enter a whole number: 5
5 is less than or equal to 5
This statement is not part of the above if else
$
```

Relational and Logical Operators

Many decisions in a programming language depend upon how one value relates to another.

The Relational Operators in Python

Table 7. Relational Operators

Operator	Meaning
<	Strictly less than
<=	Less than or equal
>	Strictly greater than
>=	Greater than or equal
==	Equal
!=	Not equal
`is`	Object identity
`is not`	Negated object identity

The Logical Operators in Python

Table 8. Logical Operators

Operator	Unary or Binary	The Result is `True` When …
`not`	unary	Operand is `False`
`and`	binary	Both operands must be `True`
`or`	binary	Only one operand needs to be `True`

- The logical operators allow expressions consisting of compound conditions such as the ones shown next.

logicals.py

```
#!/usr/bin/env python3


x = 0
y = 0

if x == 0 and y == 0:
    print('x and y are zero')

if x == 0 or y == 0:
    print('x or y or both are zero')

if not (x >= 10 and x <= 20):
    print('x not between 10 and 20')
```

The output of the above program is shown next.

```
$ python3 logicals.py
x and y are zero
x or y or both are zero
x not between 10 and 20
$
```

- Both the `and` and the `or` are *short-circuited* operators.
  - `or`: This means as soon as a condition in an or series of conditional tests evaluates to `True`, any remaining test conditions in the or series are not evaluated
  - `and`: Likewise, as soon as a condition in an and series of conditional tests evaluates to `False`, any remaining test conditions in the and series are not evaluated
- The relational and logical operators yield results of either `True` or `False`.
- In addition to the literal values `True` and `False`, there are other expressions that evaluate to `True` or `False` when used where a conditional statement is expected.

- `True`
  - Any non-zero value
- `False`
  - `0`
  - Empty sets, dictionaries, tuples, or lists
  - Empty strings
  - `None` – A built-in constant frequently used to represent the absence of a value

Python Ternary-like if Statements

A single-line if statement can be used in Python as a substitution for a multi-line if statement. This is the equivalent of a ternary operator.

- The general syntax of a single-line if statement is:
  - [true_part] if [condition] else [false_part]
  - When the given condition is True, the true_part is returned.
  - When the given condition is False, the false_part is returned.

ternary_like.py

```
#!/usr/bin/env python3


first, second = input("Enter 2 numbers separated by a space: ").split()

first = int(first)
second = int(second)

greater = first if first > second else second
print(greater)

print(first if first > second else second)

print('even' if first % 2 == 0 else 'odd')
```

The output of the above program is shown next.

```
$ python3 ternary_like.py
Enter 2 numbers separated by a space: 10 6
10
10
even
$
```

The Walrus Operator

New to Python 3.8 is the walrus operator `:=`. The walrus operator is an assignment expression. It assigns values to variables as part of an expression.

- Variables being assigned a value using the walrus operator do not need to be declared prior to the assignment expression.
- The generic syntax is `variable := expression`.
- Assignment expressions are especially useful in the header of a conditional check.
  - Part of the intent is to consolidate an assignment statement with a boolean expression.

i_am_the_walrus.py

```
#!/usr/bin/env python3


# Some artists that covered the Beatles I am the Walrus
covers = {'Spooky Tooth': 1970, 'Crack the Sky': 1978, 'Chill Faction':
1987,
          'Ian Kelly': 2012, 'Frank Zappa': 2009, 'Styx': 2005,
          'Men Without Hats': 1991, 'Oasis': 1994}

band = 'Styx'

# old school (< 3.8pw : pre-walrus)
year = covers.get(band)
if year:
    print(f'{band} covered I am the Walrus in {year}.')
else:
    print(f'{band} did not cover I am the Walrus.')


# new school with the walrus operators
if year:= covers.get(band):
    print(f'{band} covered I am the Walrus in {year}.')
else:
    print(f'{band} did not cover I am the Walrus.')

if (walrus_count := len(covers)) < 10:
    print(f'{walrus_count} is not enough I am the Walrus covers.')
```

The output of the above program is shown next.

```
$ python3 i_am_the_walrus.py
Styx covered I am the Walrus in 2005.
Styx covered I am the Walrus in 2005.
8 is not enough I am the Walrus covers.
$
```

Exercises

Exercise 1

Write a program that prompts for an integer.

- The program should verify that the number can safely be converted to an `int`.
  - If it can, multiply the number by 100 and print the result.
  - If it can't, the program should print a message indicating the value was not an integer.

Exercise 2

Write a program that prompts for a floating point number.

- The program should verify that the number can safely be converted to a `float`.
  - If it can, multiply the number by 100 and print the result.
  - If it can't, the program should print a message indicating the value was not a floating point number.

Exercise 3

Write a program that prompts the user for two numbers.

- The program should print "Low/High" if the first number was less than the second number.
- "High/Low" if the second number was less than the first.
- "Equal" if the two numbers are equal.

Exercise 4

Write a program that prompts the user for an hourly wage, and then a number of hours worked.

- The program should then print the wages to be paid.
- If more than 40 hours were worked (i.e., overtime), the employee should get 1.5 times their usual rate for every hour over 40.
- The program should work if the user enters a dollar sign as the first character in their rate or not.

Exercise 5

According to the United States constitution, a person is eligible to be a US Senator if they are at least 30 years old and have been a citizen for at least 9 years. They are eligible to be a US Representative if they are at least 25 years old and have been a US citizen for 7 years. To be president, a person must be at least 35 years old and be born as a US citizen (i.e., a citizen their entire life).

- Write a program that prompts for person's age and years of citizenship.
- The program should determine and print their eligibility for each of these important positions.

Exercise 6

Write a program that determines if a given year is a leap year or not.

- A year is a leap year if it is evenly divisible by 4.
- However, if the year is evenly divisible by 100, it is **not** a leap year.
  - Unless it's evenly divisible by 400, in which case it's a leap year again!
    - For example, 2000 was a leap year, but 1900 was not.

Chapter 12. Loops

Objectives

- Use for loops to iterate through iterables.
- Use while loops to perform repetitive operations.
- Utilize the range and enumerate functions.
- Control loop termination.

The for Loop

The for loop in Python is used to iterate over the items of any sequence, such as a string, list, dictionary, or any iterable object.

- The generic syntax of a for loop is
  - ▶ `for target in sequence: suite`
- Each item in the sequence is evaluated once and assigns the item to the target and then the suite is executed
- The loop terminates after processing the last item in the sequence.

for_loops.py

```
#!/usr/bin/env python3


word = 'Hello'

print(word)
for each_character in word:
    print(each_character, end='\t')

print()

# list loops
numbers = [10, 20, 30, 40, 50]
for number in numbers:
    print(number, end='\t')

print()

values = input('Enter some data separated by a space: ').split()
for value in values:
    print(value)
```

The output of the above program is shown below.

```
$ python3 for_loops.py
Hello
H   e   l   l   o
10  20  30  40  50
Enter some data separated by a space: this is a 7
this
is
a
7
$
```

Iterating Through a Dictionary

As mentioned earlier, dictionaries provide methods that can be used to iterate through the keys, values, or key/value pairs

- The `keys()` method returns a view of the dictionary keys.
- The `values()` method returns a view of the dictionary values.
- The `items()` method returns a view of the dictionary items.
- Each item in the view returned by `items()` is a 2-`tuple` consisting of a key and a value.
- The views returned by the above methods are iterable, enabling them to be used in a for loop to iterate through all of its members.
- In addition to the above methods, that return views, a dictionary itself is iterable that offers up each key as it is iterated over.
- The example on the following page shows the various techniques of looping through a dictionary.

dictionary_loops.py

```
#!/usr/bin/env python3


lotteries = {'Mega Millions': 227000000, 'Power Ball': 40000000,
             'Multi Match': 625000}

# loop through the keys
for key in lotteries.keys():
    print(f'{key}:${lotteries[key]:,d}', end=' ## ')

print('\n', '-' * 50)

# loop through the dictionary
for key in lotteries:
    print(f'{key}:${lotteries[key]:,d}', end=' ## ')

print('\n', '-' * 50)

# loop through the values
for amount in lotteries.values():
    print(f'${lotteries[key]:,d}', end=' ## ')

print('\n', '-' * 50)

# loop through the key/value pairs as a tuple
for item in lotteries.items():
    print(item, f'{item[0]}: ${item[1]:,d}')
```

The output of the above program is shown on the following page.

```
$ python3 dictionary_loops.py
Mega Millions:$227,000,000 ## Power Ball:$40,000,000 ## Multi
Match:$625,000 ##
 --------------------------------------------------
Mega Millions:$227,000,000 ## Power Ball:$40,000,000 ## Multi
Match:$625,000 ##
 --------------------------------------------------
$625,000 ## $625,000 ## $625,000 ##
 --------------------------------------------------
('Mega Millions', 227000000) Mega Millions: $227,000,000
('Power Ball', 40000000) Power Ball: $40,000,000
('Multi Match', 625000) Multi Match: $625,000
$
```

The range Function

A range object can also be used to represent a sequence of numbers and then iterate over the range as a sequence with a for loop.

- range(start, stop[,step])
  - The start parameter defines the number at which to start.
    - If no value is provided for the start, it will default to zero.
  - The stop parameter defines the number at which to stop.
    - The stop is exclusive, meaning the last number the range returns is one less than the value of stop.
  - The step parameter defaults to one.
    - If a value is provided for the step, it must be greater than zero and the contents of the range are determined by the formula r[i] = start + step*i where i >= 0 and r[i] < stop.
    - Negative values can be specified for the step.
    - The formula for negative steps is the same but the constraints are i >= 0 and r[i] > stop.

ranges.py

```
#!/usr/bin/env python3


print('looping through range(10):')
for i in range(10):
    print(i, end=' ')

print('\n\nlooping through range(10, 21)')
for i in range(10, 21):
    print(i, end=' ')

print('\n\nlooping through range(5, 30, 5)')
for i in range(5, 30, 5):
    print(i, end=' ')

print()
```

The output of the above program is shown below.

```
$ python3 ranges.py
looping through range(10):
0 1 2 3 4 5 6 7 8 9

looping through range(10, 21)
10 11 12 13 14 15 16 17 18 19 20

looping through range(5, 30, 5)
5 10 15 20 25
$
```

The enumerate Function

The enumerate function returns an enumerate object which is iterable. Each item returned from the enumerate object will be a tuple.

- enumerate(iterable, start=0)
  - The tuple returned will contain two things, a count and the value from the iterable object passed to the function.
  - If a value is not passed for the start parameter, it defaults to zero.
    - In this case, the tuple will contain the index position and the value at that index position from the iterable object.
- This method is useful when needing to modify items in a collection, such as a list.

enumerator.py

```
#!/usr/bin/env python3


values = [10, 20, 30, 40, 50]

print(values)

for pair in enumerate(values):
    print(type(pair), pair)

print()
for index, value in enumerate(values, start=1):
    values[index] *= 100

print
for index, value in enumerate(values):
    print(index, value)

print(values)
```

The output of the above program is shown below.

```
$ python3 enumerator.py
[10, 20, 30, 40, 50]
<class 'tuple'> (0, 10)
<class 'tuple'> (1, 20)
<class 'tuple'> (2, 30)
<class 'tuple'> (3, 40)
<class 'tuple'> (4, 50)

0 1000
1 2000
2 3000
3 4000
4 5000
[1000, 2000, 3000, 4000, 5000]
$
```

The while Loop

The while statement causes Python to loop through a suite of statements if the test expression evaluates to True.

- The while statement uses the same evaluations for True and False as the if statement.
  - Likewise, the same indentation rules are used for a while as they are for an if.
- Each time through the loop, if the condition in the while is True, then the body of the while loop is executed.
- When the condition is False, then the loop is complete and first statement after the while loop is executed.
- while loops often require the use of a counter variable.
  - When this is the case, remember to increment the counter variable to prevent an infinite loop.

while_loops.py

```
#!/usr/bin/env python3


counter = 5
total = 0

while counter <= 10:
    total += counter
    counter += 1
    print(f'Running Total = {total}, Counter = {counter}')

print()
print(f'Final Total = {total}')

numbers = [10, 20, 30, 40, 50]
counter = 0
while counter < len(numbers):
    numbers[counter] *= 10
    counter += 1

print(numbers)

while (value := input('Type something (or quit to stop:) ')) != 'quit':
    print('Value is:', value)
```

- The output of the above program is shown below.

```
$ python3 while_loops.py
Running Total = 5 Counter = 6
Running Total = 11 Counter = 7
Running Total = 18 Counter = 8
Running Total = 26 Counter = 9
Running Total = 35 Counter = 10
Running Total = 45 Counter = 11
Final Total = 45
[100, 200, 300, 400, 500]
Type something (or quit to stop:) test
Type something (or quit to stop:) 5
Type something (or quit to stop:) quit
Value is: test
Value is: 5
$
```

break and continue

Any looping construct can have its control flow changed through a break or continue statement within it.

- When a break is executed, control of the program jumps to the first statement beyond the loop.
- A break is often used when searching through a collection for the occurrence of a particular item.
- When a continue statement is executed, the rest of the suite is skipped for that iteration of the loop and control goes to the next iteration.

continue_and_break.py

```
#!/usr/bin/env python3
value = 0
while value < 100:
    value += 1
    if value % 2 != 0:
        print(f'{value} is odd.')
        continue
    if value == 10:
        break
    print(f'{value}')
```

- The output of the above program is shown next.

```
$ python3 continue_and_break.py
1 is odd.
2
3 is odd.
4
5 is odd.
6
7 is odd.
8
9 is odd.
$
```

Loops With else

Both for loops and while loops can have an optional else clause.

- The else clause suite will execute when a loop is terminated normally.
- The else clause suite will not execute when the loop terminates as the result of a break statement.

while_with_else.py

```
#!/usr/bin/env python3


counter = 1
total = 0
while counter <= 10:
    total += counter
    counter += 1
else:
    print('Total is:', total)
```

- The output of the above program is shown next.

```
$ python3 while_with_else.py
Total is: 55
$
```

Exercises

Exercise 1

Write a program that inputs a character, and then a number, and then prints that many of the character using a loop.

```
$ Enter a character: *
$ Enter a number: 5
*****
```

- If the user enters a string instead of a character, print as many copies of the string as possible without printing more total characters than the number entered.

```
$ Enter a character: trololo
$ Enter a number: 23
$ trololotrololotrololo
```

Exercise 2

Write a program that prompts twice for an integer.

- The program should output the sum of the integers within the range of those two numbers inclusively.
- For example, if the user inputs the numbers *10* and *15*, then the sum would be *75*.

```
10 + 11 + 12 + 13 + 14 + 15 = 75
```

Exercise 3

In mathematics, the Collatz sequence is generated by starting with any positive integer and repeatedly applying the following functions until the value 1 is reached:

$Collatz(N_{even}) = N / 2$
$Collatz(N_{odd}) = 3N + 1$

- Although no exceptions have been found, there is no formal mathematical proof as to whether or not this sequence always ends in 1.
- Write a program that inputs a positive integer from the user and prints all the values in the Collatz sequence for that integer.

Exercise 4

Create a program that mimics NASA's final 31 second countdown to liftoff. The sequence is:

- Ground launch sequencer is go for auto sequence start (T-31 seconds)
- Activate launch pad sound suppression system (T-16 seconds)
- Activate main engine hydrogen burnoff system (T-10 seconds)
- Main engine start (T-6.6 seconds)
- T-0 Solid rocket booster ignition and liftoff!

Exercise 5

Use a loop through to print each number from 0 to 49 to produce the following output.

- Each number should be printed individually as opposed to concatenating them as a string.

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
```

Exercise 6

Write a program that inputs a phrase from the user, and outputs the corresponding acronym (i.e. the capitalized first letter of each word of the phrase).

Exercise 7

Write a program that inputs a sentence from the user, creates a list of words from the sentence and then creates a list of word lengths. Output the list of words and their lengths.

Exercise 8

Write a program that prompts the user to enter some words. Store the words in a `list`. Then prompt the user for more words and store them in another `list`. Print out all the in the first `list` that are not in the second `list`.

Exercise 9

Prompt a user to enter a sentence.

- Create a program that contains a dictionary of each character in the string as the key and a count of the number of times the character is in the sentence as the value.
- The program should print the character and the character count, sorted by the characters with the highest count first.

Exercise 10

Write a program that prompts the user to enter a positive integer, N, then outputs N lines each with N stars.

Exercise 11

Write a program that has a list of numbers.

- Loop through the numbers and modify each element to be itself times its index position.
- Print the resulting list

Chapter 13. Lab 02

Objectives

- Gain a better understanding of previously covered concepts of the Python programming language through the completion of practical exercises.

Exercises

Exercise 1

Write a program that prompts for a user's first and last name.

- The program should print out the person's full name on one line and initials on another line.

Exercise 2

Given the following three book titles as literal strings. Include them in a program that will convert them to the correct case for a title, as shown in the comment to the right of each.

```
book1 = "gone with the wind"                      # Gone with the Wind
book2 = "Harry potter And the sorcerer's Stone"  # Harry Potter and the
Sorcerer's Stone
book3 = "The lIOn, ThE witch and THE wardrobe"   # The Lion, the Witch and
the Wardrobe
```

Exercise 3

Write a program that prompts the user for a word and then prompts again for a whole number.

- The program should create a single new string of the word repeated the number of times given and print out the string.

Exercise 4

Write a program that prompts the user to enter a string of text.

- The program should print out the number of each of the vowels found in the string.

Exercise 5

Write a program that asks the user to enter a palindrome.

- The program should print out whether it is true or false that the word is a palindrome.
  - ▶ Palindromes are words that are the same when spelled backwards such as 'racecar'.

Exercise 6

Write a program that calculates and prints the index of every occurrence of the letter "i" in Mississippi.

Exercise 7

Write a program that asks the user to type a string.

- The program should display the words with a border around it.
  - The border can be any character of your choosing.
- For example: If the user enters "Happy Birthday", the output should look similar to:

```
##################
# Happy Birthday #
##################
```

Exercise 8

Write a program that prompts the user twice for a number.

- Print each number on its own line.
- Print the result of multiplying the two numbers together.
- Print the number of characters in the result of multiplying the two numbers together.

Exercise 9

Write a program that prompts the user for a word with an odd number of characters in it. Validate that the word has an odd number of characters. If it does not, continually prompt the user for a word with an odd number of characters until one is entered.

- The program should then calculate the index of the character in the middle of the word and print that character and its index.

Exercise 10

Write a program that prompts the user for a word with at least 9 characters. Continually prompt the user until a word with at least that length is entered.

- The program should print the word without the first 2 and last 2 characters
  - For example "Something" should result in "methi" in the display.

Exercise 11

Write a program that prompts once for the user to supply 3 numbers.

- Print out the three numbers and which of the 3 is the largest

Exercise 12

Write a program that prompts the user to enter some text.

- The program should prompt twice for the user to enter some text.
  - Combine all of the words from both inputs into a single list and display the contents of the combined list.

Exercise 13

Rewrite the previous exercise that prompts for two inputs of text from the user.

- The program should display the sorted contents of the list.
  - The output should be the words joined together with a space between each word in the output.

Exercise 14

Write a program that unpacks the following list into variables.

- Print the value of two of the variables on one line.
- Print the value of the other two variables on a second line.

```
data = [[1, 3, 7], [2, 4, 6], ["a", "b", "c"], ["x", "y", "z"]]
```

Exercise 15

Given the following list of numbers:

```
numbers = [10, 16, 13, 10, 14, 17, 11, 18, 12]
```

- Write a program that sorts the above list of numbers and prints them out.
- The program should then use the numbers to determine which position to insert the number 15 into so that the contents of the list remain sorted.
  - ▶ Insert the 15 into the calculate index and reprint the contents of the list.

Exercise 16

Write a program that creates a list of fruits and prompts the user to choose one from the list.

- The program should remove the chosen fruit from the list and print the remaining fruits.

Exercise 17

Write a program that prompts the user for several names separated by spaces

- Print out the longest name.

Exercise 18

Write a program that creates two lists of numbers.

- Calculate and print out the largest number in each of the two lists.
- Calculate and print out the largest number in the two lists combined.

Exercise 19

Write a program that prompts the user 4 times to enter some text and collect all four inputs in a list.

- Determine from the list if any of the user input was simply the <return> key being entered.
  - ▶ If so, print that an invalid entry was found and the index position at which it was found.

Exercise 20

Write a program that defines a dictionary of five states and their capitals.

- The program should prompt the user to supply the capital of a given state from the

dictionary.

- The program should then print whether it is true or false that their choice is the capital.

Exercise 21

Rewrite the previous exercise that works with states and their capitals.

- The program should still define a dictionary of 5 states and capitals.
- It should then display the states in the dictionary and prompt the user to enter a missing state and its capital.
  - Prove that adding it to the dictionary worked by getting the value of the new state from the dictionary and print it out.

Exercise 22

Create a program that stores a dictionary of employee names and a list of days of the week that employee can work.

- The program should define the days of the week only once and then reference them as many times as needed.
- The dictionary should have at least four employees in it and each employee must work at least two days.
  - How the data is supplied is entirely up to you.
- Prompt the user for whose schedule they would like to see and print out that employee and their schedule.

Exercise 23

Write a program that asks the user what time it is in the form of HH:MM (24 hour clock)

- Respond in the output with either one of:
  - "Wow, that’s early" for the hours from 0 - 6
  - "Good morning" for the hours of 7 - 11
  - "Good afternoon" for the hours of 12 - 17
  - "Good evening" for the hours of 18 - 23

Exercise 24

Write a program that prompt the user for a positive whole number:

- Print one message if the number given is positive and even
- Print a different message if the number is positive and odd
- Print a different message if it is not ether of the first two.

Exercise 25

Write a program that loops, prompting for the user to enter some numbers.

- The program should keep running total of all of the numbers, keeping in mind the user might enter several numbers on the same line.
- The program should continue prompting until the user types "done".
- The program should then print out the sum of all the numbers supplied.

Exercise 26

Rewrite the above program so that the loop stops when the user enters nothing.

Exercise 27

Write a program that prompts the user to enter some text.

- Loop through the input and determine how many characters are upper case and how many are lower case.

Exercise 28

Write a program that defines a dictionary of the months and how many days are in the month. The program will need to properly handle leap years.

- Print out the contents of the dictionary with each line being a month and its number of days.
- Print out the contents sorted by the number of days in the month from highest to lowest.
- Print out the contents of the dictionary using the 3 character short name for each month
    - ▶ Make sure that each printout of the dictionary is separated clearly from the next printout of the dictionary

Exercise 29

Write a program that prints out the circumference and area of circles.

- The value 3.14 can be used for π
- The circumference of a circle is 2πr where r is the radius.
- The area of a circle is πr^2

The program should prompt for three values from the user:

- A starting radius.
- An ending radius.
- And an increment value.
  - So for example if the user entered the three numbers: 10 20 2
  - The program should print out the circumference and area for circles with a radius of 10, 12, 14, 16, 18 , and 20
    - Notice that the ending value should be inclusive.

Exercise 30

Rewrite the program that had a dictionary of the 12 months as a program that has two lists.

- One list contains the names of the months.
- The other list contains the number days in each month.
  - Write a loop that process the two lists in parallel.
  - Parallel in the sense that what is in index 0 of one is related to index 0 of the other.
- The loop should print the month name in the first column and the number of days in the second column.
  - The data will need to be formatted so that everything lines up in columns.

Chapter 14. Review

Objectives

- Review Intermediate Strings
- Review Lists
- Review Dictionaries
- Review Conditionals
- Review Loops

Intermediate Strings

Strings in Python are represented as the data type `str` and represent an immutable sequence of zero or more characters.

Literal Strings are values that are specified inside of quotes.

- The quotes can be a pair of single or double quotes
  - Literal multi-line strings use pairs of 3 single or double quotes.
  - Multiple lines within a literal string can also be accomplished using the `\` as the line continuation character
- Prefixing a literal string with an `r` or an `R` makes it a raw string where no escaping is done within the string.

The `str` class has an abundance of methods that can be called on instances of strings.

- Some of the methods return a Boolean value, while other methods of the class return a different string.
  - The methods that return a string do so because the original string is immutable.

The example that follows uses the various methods of the `str` class that modify the casing (upper/lower) of characters in the string they are called on.

modifying_case.py

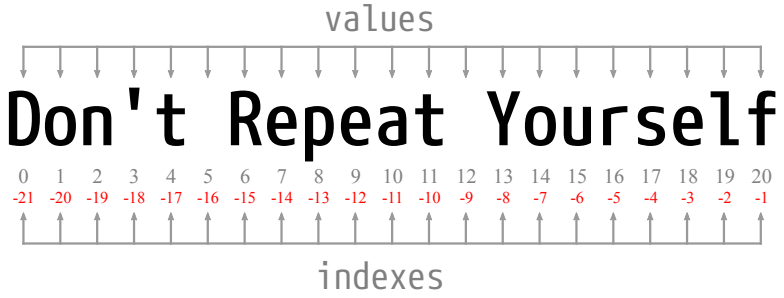
```
#!/usr/bin/env python3

# Converting to upper and lower case:
data = "Her name is Helen and she is from MD"
lower_case = data.lower()
upper_case = data.upper()
print(f"Original:{data}\nUpperCase:{lower_case}\nLowerCase:{upper_case}\n"
)

# casefold can be used to compare unicode strings in different languages:
english = "gross"
german = "groß"
print("Lower case: groß == gross ?", english.lower() == german.lower())
print("Case fold: groß == gross ?", english.casefold() ==
german.casefold())
print()

# Title case
print("Title Case:", data.title())

# Capitalize
print("Capitalize:", data.capitalize())
```

```
$ python3 modifying_case.py
Original:Her name is Helen and she is from MD
UpperCase:her name is helen and she is from md
LowerCase:HER NAME IS HELEN AND SHE IS FROM MD

Lower case: groß == gross ? False
Case fold: groß == gross ? True

Title Case: Her Name Is Helen And She Is From Md

Capitalize: Her name is helen and she is from md
$
```

Strings are sequence types and as such can use index and slicing syntax to obtain a sub-string of the string the operation is performed on.

- Indexing involves a single character at the specified index position.
- Slicing involves two integers separated by a colon in the square brackets.
- Extended Slicing involves three integers separated by colons.

The image below shows the value for each index of the given string.



- The string is used in the following example, demonstrating using slicing and indexing to obtain sub-strings from a given string.

string_indexing.py

```
#!/usr/bin/env python3

# Creating an acronym
words = "Don't Repeat Yourself"
first_break = words.find(' ')
second_break = words.find(' ', first_break + 1)
acronym = f"{words[0]}{words[first_break + 1]}{words[second_break + 1]}"

print("Spaces found at indexes", first_break, "and", second_break)
print(acronym, ":", words)
```

```
$ python3 string_indexing.py
Spaces found at indexes 5 and 12
DRY : Don't Repeat Yourself
$
```

The next example uses slicing and extended slicing.

string_slicing.py

```
#!/usr/bin/env python3

# Creating an acronym
words = "Don't Repeat Yourself"
first_break = words.find(' ')
second_break = words.find(' ', first_break + 1)

# Slicing to get each word
print(words[:first_break])
print(words[first_break + 1:second_break])
print(words[second_break + 1:])
print()

# Using negative indexes
print(words[-21:-16])
print(words[-15:-9])
print(words[-8:])
print()

# The making of an Emordnilap (Palindrome backwards)
word = "desserts"
print(word, ":", word[::-1])
word = "drawer"
print(word, ":", word[::-1])
```

```
$ python3 string_slicing.py
Don't
Repeat
Yourself

Don't
Repeat
Yourself

desserts : stressed
drawer : reward
$
```

Lists

Whereas a string is a sequence of characters, a `list` in Python is a sequence of arbitrary objects.

- Since a `list` is a sequence, it is able to use the same index and slice notation introduced for strings.

A `list` is an ordered collection, meaning its values can be retrieved by index, of zero or more objects.

- A `list` is dynamic in that it size changes as objects are added and/or removed.

A `list` can be created using the `list()` constructor or a comma separated listed of zero or more objects inside of square brackets.

The example that follows demonstrates the creating, indexing, slicing, and unpacking of lists.

woods.py

```
#!/usr/bin/env python3

hardwoods = ["Oak", "Ash", "Beech", "Elm", "Mahogany", "Teak", "Walnut"]
softwoods = []
softwoods.append("Pine")
softwoods.append("Cedar")
softwoods.append("Fir")
wood_types = softwoods + hardwoods

print(f"There are {len(hardwoods)} hardwoods and {len(softwoods)}
softwoods")
print("One of the softwoods is", softwoods[2])
print("Some of the hardwoods are", ", ".join(hardwoods[1::2]))
print("A list of all of the wood types are as follows:")
print("\n".join(wood_types))
print()

# Unpacking the softwoods
pine, cedar, fir = softwoods
fmt = "{:>5}:"
print(fmt.format(cedar), "used for everything from house siding to
guitars")
print(fmt.format(fir), "used frequently for veneers and plywood")
print(fmt.format(pine), "widely used in furniture")
```

```
$ python3 woods.py
There are 7 hardwoods and 3 softwoods
One of the softwoods is Fir
Some of the hardwoods are Ash, Elm, Teak
A list of all of the wood types are as follows:
Pine
Cedar
Fir
Oak
Ash
Beech
Elm
Mahogany
Teak
Walnut

Cedar: used for everything from house siding to guitars
  Fir: used frequently for veneers and plywood
 Pine: widely used in furniture
$
```

The next example demonstrates the dynamic aspect of lists by adding, updating, and removing elements of the list.

board_games.py

```
#!/usr/bin/env python3

board_games = ["Sorry", "Yahtzee", "Risk", "Monopoly", "Clue", "Mille
Bornes"]
print("Board Games:", ", ".join(board_games))

# Remove Yahtzee and Mille Bornes since they are not board games.
print(board_games.pop(), "is a Card Game more than a Board Game.")
print(board_games.pop(1), "is a Dice Game more than a Board Game.")
print()
print("Corrected Board Games:", ", ".join(board_games))

"Which of the board games would you like to play?"
fmt = "{}: {}"
print(fmt.format(1, board_games[0]))
print(fmt.format(2, board_games[1]))
print(fmt.format(3, board_games[2]))
print(fmt.format(4, board_games[3]))

choice = input("Please enter a number from above: ")
print()
print("You have chosen ", board_games.pop(int(choice)))
print("This leaves ", ", ".join(board_games), "for others to choose
from.")

more = ["Stratego", "Othello", "Mousetrap", "Concentration"]
print("The following games will be added to the list:")
print(more)
board_games.extend(more)

new_game = input("Provide another board game you are familiar with: ")
board_games.insert(0, new_game)
print("Current list of board games\n", ", ".join(board_games), sep='')
```

The output of the above program is shown on the following page.

```
$ python3 board_games.py
Board Games: Sorry, Yahtzee, Risk, Monopoly, Clue, Mille Bornes
Mille Bornes is a Card Game more than a Board Game.
Yahtzee is a Dice Game more than a Board Game.

Corrected Board Games: Sorry, Risk, Monopoly, Clue
1: Sorry
2: Risk
3: Monopoly
4: Clue
Please enter a number from above: 2

You have chosen  Monopoly
This leaves  Sorry, Risk, Clue for others to choose from.
The following games will be added to the list:
['Stratego', 'Othello', 'Mousetrap', 'Concentration']
Provide another board game you are familiar with: Candy Land
Current list of board games
Candy Land, Sorry, Risk, Clue, Stratego, Othello, Mousetrap, Concentration
$
```

Lists can be sorted with either the `sort()` method of the `list` class or the built-in `sorted()` function.

- The `sort()` method sorts a list in place (modifies the contents of the `list` itself)
- The `sorted()` funtion leaves the original list intact and instead returns a new list with its contents sorted.

sorting_basics.py

```
#!/usr/bin/env python3
board_games = ["Sorry", "Risk", "Clue", "Stratego", "Othello",
               "Mousetrap", "Concentration"]
fmt = "{:<20}{}"
sep = ", "
print(fmt.format("Original:", sep.join(board_games)))
# Return and print a sorted copy of the board games
# in ascending and descending order
print(fmt.format("A-Z:", sep.join(sorted(board_games))))
print(fmt.format("Z-A:", sep.join(sorted(board_games, reverse=True))))

# The sorted() function leaves the original list unchanged
print(fmt.format("Original:", sep.join(board_games)))

# Whereas the sort() method changes the actual list.
board_games.sort()
print(fmt.format("Original now A-Z:", sep.join(board_games)))
print()

# Mixed case does tno necessarily sort alphabetically
board_games[0] = board_games[0].lower()
board_games[5] = board_games[5].lower()
print(fmt.format("Current Games:", sep.join(board_games)))
print(fmt.format("Sorted Games:", sep.join(sorted(board_games))))
```

```
$ python3 sorting_basics.py
Original:           Sorry, Risk, Clue, Stratego, Othello, Mousetrap,
Concentration
A-Z:                Clue, Concentration, Mousetrap, Othello, Risk, Sorry,
Stratego
Z-A:                Stratego, Sorry, Risk, Othello, Mousetrap,
Concentration, Clue
Original:           Sorry, Risk, Clue, Stratego, Othello, Mousetrap,
Concentration
Original now A-Z:   Clue, Concentration, Mousetrap, Othello, Risk, Sorry,
Stratego

Current Games:      clue, Concentration, Mousetrap, Othello, Risk, sorry,
Stratego
Sorted Games:       Concentration, Mousetrap, Othello, Risk, Stratego,
clue, sorry
$
```

Custom sorting can be achieved by passing the named argument `key` with an appropriate value as an optional argument to either the `sort()` method of the `list` or the `sorted()` function.

- The value of the `key` argument should be a reference to a single argument function.
  - The return value of the function will then be used as the comparison key when sorting each element in the list.

The next example demonstrates the use of the `str.lower` method to sort in a case insensitive manner.

case_insensitive.py

```
#!/usr/bin/env python3
board_games = ["sorry", "Risk", "Clue", "Stratego", "othello",
               "Mousetrap", "concentration"]
fmt = "{:<15}{}"
sep = ", "

print(fmt.format("Original:", sep.join(board_games)))

# Sort using a reference to the lower function of the str class
board_games.sort(key=str.lower)
print(fmt.format("Sorted Games:", sep.join(board_games)))
```

```
$ python3 case_insensitive.py
Original:      sorry, Risk, Clue, Stratego, othello, Mousetrap,
concentration
Sorted Games:  Clue, concentration, Mousetrap, othello, Risk, sorry,
Stratego
$
```

Sorting a list would provide one way of determining the smallest or largest element in a list.

- A more Pythonic way of achieving this would be to rely upon some of the built-in Python functions that work with lists and provide these types of calculations.

The example below demonstrates the min and max functions.

min_max.py

```
#!/usr/bin/env python3
hardwoods = ["Oak", "Ash", "Beech", "Elm", "Mahogany", "Teak", "Walnut"]
softwoods = ["Pine", "Cedar", "Fir"]
wood_types = softwoods + hardwoods

print("The 'minimum' wood is", min(wood_types))
print("While the 'maximum' wood is", max(wood_types))
```

```
$ python3 min_max.py
The 'minimum' wood is Ash
While the 'maximum' wood is Walnut
$
```

Similar to the optional `key` argument in regards to sorting, the min() and max() functions also have an optional `key` argument that is used as the comparison key as to whether one element is larger or smaller than another element in the collection passed to it.

The next example will use the `len` function to determine the length of the longest word in the given list in order to use it in the string formatting of the output.

longest.py

```
#!/usr/bin/env python3
hardwoods = ["Oak", "Ash", "Beech", "Elm", "Mahogany", "Teak", "Walnut"]

longest_hardwood = max(hardwoods, key=len)
width = len(longest_hardwood)

fmt = "{0:{1}}{2:^7}"
print(fmt.format("Name", width, "# Chars"))
print(fmt.format(hardwoods[0], width, len(hardwoods[0])))
print(fmt.format(hardwoods[1], width, len(hardwoods[1])))
print(fmt.format(hardwoods[2], width, len(hardwoods[2])))
print(fmt.format(hardwoods[3], width, len(hardwoods[3])))
print(fmt.format(hardwoods[4], width, len(hardwoods[4])))
print(fmt.format(hardwoods[5], width, len(hardwoods[5])))
print(fmt.format(hardwoods[6], width, len(hardwoods[6])))
```

```
$ python3 longest.py
Name    # Chars
Oak        3
Ash        3
Beech      5
Elm        3
Mahogany   8
Teak       4
Walnut     6
$
```

The `in` operator and the `any()` and `all()` functions can be used to determine certain details bout the contents of a collection.

The following example prompts for numbers and checks to see if any or all of the choices are in a predetermined list of numbers.

in_any_and_all.py

```
#!/usr/bin/env python3
number_pool = [2, 5, 9, 3, 19, 16, 15, 8, 14, 10]
print("Pick 3 numbers between 1 and 20")
picks = input("Choices: ").split()
results = []
results.append(int(picks[0]) in number_pool)
results.append(int(picks[1]) in number_pool)
results.append(int(picks[2]) in number_pool)
print("It is", any(results), "that any of your 3 choices are in",
number_pool)
print("It is", all(results), "that all 3 of your choices are in",
number_pool)
```

```
$ python3 in_any_and_all.py
Pick 3 numbers between 1 and 20
Choices: 3 7 12
It is True that any of your 3 choices are in [2, 5, 9, 3, 19, 16, 15, 8,
14, 10]
It is False that all 3 of your choices are in [2, 5, 9, 3, 19, 16, 15, 8,
14, 10]
$
```

Dictionaries

Dictionaries are key/value pairs where the keys have to be unique and hashable.

- They are designed to provide fast lookups to retrieve a value by its key.
- Subscript notation can be used to add key/value pairs or retrieve a value for an existing key.

The example below demonstrates creating, adding to, and retrieving information from a dictionary.

favorites.py

```
#!/usr/bin/env python3
a_key = "season"
favorites = {"color": "green", "number": 13, a_key: "Fall"}
favorites["food"] = "lasagna"
favorites["day"] = "Saturday"

print(favorites)
print("Key:", a_key, "\tValue:", favorites["season"])
print(favorites.get("number", 7))
print(favorites.get("car", "Make and Model Unknown"))
favorites["color"] = "blue"

print(favorites)
```

```
$ python3 favorites.py
{'color': 'green', 'number': 13, 'season': 'Fall', 'food': 'lasagna',
'day': 'Saturday'}
Key: season Value Fall
13
Make and Model Unknown
{'color': 'blue', 'number': 13, 'season': 'Fall', 'food': 'lasagna',
'day': 'Saturday'}
$
```

The previous example of favorites would work for storing a single person's favorites.

- To store the favorites for more than one person, a dictionary of dictionaries may be a good choice.
  - The following example uses a person's name as the key and each value will be a dictionary of favorites for that person.

peoples_favorites.py

```
#!/usr/bin/env python3
color, number, season, food, day = ["color", "number", "season", "food",
"day"]
defaults = {color: "White", number: 0, season: "Spring", food: "burger",
            day: "Friday"}


people = {"Pedro": {color: "red", number:5, season:"Summer",food:
"pizza"},
          "Alysha": {season: "Fall", color: "green", food: "lasagna",
                     day:"Friday", number:21},
          "Sunita": {season:"Summer"},
          "Adam": {number:4, color:"black"}}

# Print out the names indented from the header "Names:"
names = people.keys()
print("Names:")
names = '\n    '.join(names)
print(' ' * 3, names)
print()

# Print out the favorites indented from the header "Favorites:"
favorites = defaults.keys()
print("Favorites:")
favorites = '\n    '.join(favorites)
print(' ' * 3, favorites)
print()

print("Choose a Person and a favorite you would like to find from the
lists above")
choices = input(">")

name, favorite = choices.split()
persons_favorites = people.get(name, "John/Jane Doe")
print(f"{name}'s favorite {favorite} is", end=" ")
print(persons_favorites.get(favorite, defaults[favorite]))
```

```
$ python3 peoples_favorites.py
Names:
    Pedro
    Alysha
    Sunita
    Adam

Favorites:
    color
    number
    season
    food
    day

Choose a Person and a favorite you would like to find from the lists above
>Alysha season
Alysha's favorite season is Fall
$
```

A local club has decided to keep track of how many raffle tickets each member has sold for a fundraiser by storing the information in a dictionary.

- The next example demonstrates updating the dictionary being used with the sales from two of the club members.
  - One of the members has already sold some and as such exists in the dictionary.
  - The other member has sold some for the first time, so will be a new entry in the dictionary.
  - The designer of the dictionary decided it was better to only store those names that have sold tickets as opposed to listing every member.
- It is important to use the `get()` method with a meaningful default value for unseen keys.

raffle.py

```
#!/usr/bin/env python3
# Existing sales of raffle tickets
raffle = {"Calvin": 15, "Susan": 7, "Micky": 4}

# Calvin has sold an additional 5 tickets
raffle["Calvin"] = raffle.get("Calvin", 0) + 5

# Jeremy has sold his first 3
raffle["Jeremy"] = raffle.get("Jeremy", 0) + 3

# Current sales
print(raffle)
```

```
$ python3 raffle.py
{'Calvin': 20, 'Susan': 7, 'Micky': 4, 'Jeremy': 3}
$
```

The next example creates a dictionary of antonyms and prompts the user for the opposite of one of the key value pairs removed from the dictionary.

- It then prints the keys sorted by values and verifies by printing the sorted values also.

antonyms.py

```
#!/usr/bin/env python3
antonyms = {"Hot": "Cold", "Easy": "Hard", "Late": "Early", "True":
"False",
            "Day": "Night", "Inside": "Outside", "Above": "Below"}

# Obtain a list of the keys in the dictionary

# This is a dict_keys object which will be more helpful with loops later
keys = antonyms.keys()

# For now, convert it to a list which we know how to work with.
keys = list(keys)

# Unpack one of the items popped off (in LIFO order) of the dictionary
key, opposite = antonyms.popitem()

guess = input(f"What is the opposite of {key}? ")

print("Your answer is", opposite == guess.capitalize())
print("The opposite of", key, "is", opposite)
print()

# Display dictionary keys sorted by values
sorted_keys = sorted(antonyms.keys(), key=antonyms.get)
print(", ".join(sorted_keys))

# Match up against sorted values
sorted_values = sorted(antonyms.values())
print(", ".join(sorted_values))
```

```
$ python3 antonyms.py
What is the opposite of Above? below
Your answer is True
The opposite of Above is Below

Hot, Late, True, Easy, Day, Inside
Cold, Early, False, Hard, Night, Outside
$
```

Conditionals

Controlling the flow of execution within an application is done using the various conditional execution options that Python provides.

The following example demonstrates something that is guaranteed to happen, but conditionally something might need to be done first.

- This is a good candidate for a conditional if statement.
  - ▶ In the example a person is guaranteed to be going outside, it is just that if it is raining, an umbrella will be needed.

is_it_raining.py

```
#!/usr/bin/env python3

# Day 1
forecast = "Sunny"
print("Preparing to go outside")
if forecast == "Rainy":
    print("Getting my umbrella")
print("It is", forecast, "outside")
print()
# Day 2 - forecast has changed
forecast = "Rainy"
print("Preparing to go outside again")
if forecast == "Rainy":
    print("Getting my umbrella")
print("It is", forecast, "outside")
```

```
$ python3 is_it_raining.py
Preparing to go outside
It is Sunny outside

Preparing to go outside again
Getting my umbrella
It is Rainy outside
$
```

Often times it is necessary to test for more than one condition to know what to do next as shown in the following example.

weather_varies.py

```
#!/usr/bin/env python3
forecast = input("Is it supposed to be Sunny, Rainy, Cloudy, or Snowy
today? ")
print("Preparing to go outside")
if forecast == "Rainy":
    print("Getting my umbrella")
elif forecast == "Sunny":
    print("Getting my sun screen")
elif forecast == "Snowy":
    print("Getting my coat")
else:
    print("Guess I'll just press my luck today")
print("It is", forecast, "outside")
print()
```

The output below shows the results of running the program two times with variations on the weather forecast.

```
$ python3 weather_varies.py
Is it supposed to be Sunny, Rainy, Cloudy, or Snowy today? Snowy
Preparing to go outside
Getting my coat
It is Snowy outside

$ python3 weather_varies.py
Is it supposed to be Sunny, Rainy, Cloudy, or Snowy today? Cloudy
Preparing to go outside
Guess I'll just press my luck today
It is Cloudy outside

$
```

Relational and logical operators allow the building of more complex conditional statements that are multiple conditions combined into a single result of `True` or `False`.

The example below makes decisions based on what part of the month it is and whether there is cash on hand.

budgeting.py

```
#!/usr/bin/env python3
today = int(input("What is the day of the month? "))
cash = input("Do I have any cash? (Y/N)")[0].upper()
print()

print("I think today I should ")

if today <= 5 and cash == "Y":
    # Beginning of month with cash on hand
    print("pay my bills.")
elif today > 5 and today <= 20:
    # Middle of the month
    if cash =="Y":
        # Cash on hand
        print("buy grocercies")
    else:
        # No cash on hand
        print("plan out meals better for rest of month")
elif today > 20 and cash == "Y":
        # End of month with cash on hand
        print("spend frivolously with money left over at end of month")

else:
    print("play it safe and watch where my money goes for now")
```

```
$ python3 budgeting.py
What is the day of the month? 4
Do I have any cash? (Y/N)Y

I think today I should
pay my bills.
$
```

```
The walrus operator (introduced in Python 3.8) allows assignment of a
variable within the conditional statement
as shown in the following example.
```

conditional_assignment.py

```
#!/usr/bin/env python3
if (result := len(input("Enter a word with 5 or more letters: "))) >= 5:
    print("Your input of ", result, "characters meets the requirements")
else:
    print("The input you provided does not meet the requirements")
```

```
$ python3 conditional_assignment.py
Enter a word with 5 or more letters: hello
Your input of  5 characters meets the requirements
$ python3 conditional_assignment.py
Enter a word with 5 or more letters: abc
The input you provided does not meet the requirements
$
```

Loops

When code is repeated within an application, one way to avoid the repetition is to do the work inside of a loop.

- The two looping constructs in Python are a `for` loop and a `while` loop.
  - A `for` loop iterates over an object that is iterable such as a `str`, `list`, or `dict`
  - A `while` loop repeats as long as a specified condition evaluates to `True`.

The following example demonstrates looping through a string, a list, and a dictionary.

- The elements of a dictionary can be looped through with a for loop in various ways as demonstrated also in the example.

for_looping.py

```
#!/usr/bin/env python3

# Loopinng through a string
#     choice of 'char' as variable name is completely arbitrary
#     but as with all variable names - should be meaninful/readable
for char in "Something":
    print(char, end="*")
print('\n')

# Looping through a list of words - using a nested for loop:
sentence = "This sentence has several words in it."
for word in sentence.split():
    for char in word:
        print(char, end="*")
    print()
print()


# Looping through dictionary itself (loops through keys automatically)
antonyms = {"Hot": "Cold", "Easy": "Hard", "Late": "Early", "True":
"False",
             "Day": "Night", "Inside": "Outside", "Above": "Below"}

# item_count will be ued to print up to four items per line
item_count = 1
for key in antonyms:
    print(f"{key:>7}:{antonyms[key]:<7}", end="  ")
    if item_count % 4 == 0:
        print()
    item_count += 1
print()
```

```
$ python3 for_looping.py
S*o*m*e*t*h*i*n*g*

T*h*i*s*
s*e*n*t*e*n*c*e*
h*a*s*
s*e*v*e*r*a*l*
w*o*r*d*s*
i*n*
i*t*.*

    Hot:Cold        Easy:Hard        Late:Early       True:False
    Day:Night     Inside:Outside    Above:Below
$
```

The `keys()`, `values()`, and `items()` methods of a dictionary all return an object that is iterable and as such can be used in a for loop as shown in the following example.

keys_values_items.py

```
#!/usr/bin/env python3

hardwoods = ["Oak", "Ash", "Beech", "Elm", "Mahogany", "Teak", "Walnut"]
softwoods = ["Pine", "Cedar", "Fir"]
darkwoods = ["Mahogany", "Walnut", "Ebony"]
lightwoods = ["Maple", "Ash", "Apple", "Beech"]
wood_types = {"Hard": hardwoods, "Soft": softwoods,
              "Dark": darkwoods, "Light": lightwoods}

# Looping by keys:
print("Looping by keys")
for wood_type in wood_types.keys():
    print(wood_type, " : ", wood_types[wood_type])
print()

# Looping by values:
print("Looping by values")
for wood_collection in wood_types.values():
    print(wood_collection)
print()

# Looping by items - unpacking each tuple into a type and collection:
print("Looping by items")
for type, collection in wood_types.items():
    print(type, " : ", collection)
print()


# Using a nested for loop to loop through a dictionary of Lists
for type, collection in wood_types.items():
    print(type,":", end = "")
    for name in collection:
        print(name, end=" ")
    print()
print()
```

```
$ python3 keys_values_items.py
Looping by keys
Hard  :  ['Oak', 'Ash', 'Beech', 'Elm', 'Mahogany', 'Teak', 'Walnut']
Soft  :  ['Pine', 'Cedar', 'Fir']
Dark  :  ['Mahogany', 'Walnut', 'Ebony']
Light  :  ['Maple', 'Ash', 'Apple', 'Beech']

Looping by keys
['Oak', 'Ash', 'Beech', 'Elm', 'Mahogany', 'Teak', 'Walnut']
['Pine', 'Cedar', 'Fir']
['Mahogany', 'Walnut', 'Ebony']
['Maple', 'Ash', 'Apple', 'Beech']

Looping by items
Hard  :  ['Oak', 'Ash', 'Beech', 'Elm', 'Mahogany', 'Teak', 'Walnut']
Soft  :  ['Pine', 'Cedar', 'Fir']
Dark  :  ['Mahogany', 'Walnut', 'Ebony']
Light  :  ['Maple', 'Ash', 'Apple', 'Beech']

Hard :Oak Ash Beech Elm Mahogany Teak Walnut
Soft :Pine Cedar Fir
Dark :Mahogany Walnut Ebony
Light :Maple Ash Apple Beech
$
```

`range` and `enumerate` provide additional capabilities to looping through iterable objects since they themselves are iterable also.

`range()` can be passed one to three arguments.

- `range(10)` will start at 0 and end at 9 since the argument given as to where to stop is exclusive
- `range(5, 10)` will start at 5 and end at 9 since the second argument given as to where to stop is exclusive.
- `range(10, 5, -1)` will start at 10 and end at 6, since the second argument is exclusive, and count down by -1.

The following examples uses nested for loops and `range` to output a multiplication table up to a

specified limit.

multiplication_table.py

```
#!/usr/bin/env python3

# Print column headers
limit = 12
exclusive_limit = limit + 1
for row in range(exclusive_limit):
    if row == 0:
        row = ""
    print(f"{row:^5}", end="")
print()
print("-" * 5 * exclusive_limit)

# Print rest of table
for row in range(1, exclusive_limit):
    # print row header
    print(f"{row:^3} |", end="")

    for column in range(1, exclusive_limit):
        print(f"{row * column:^5}",  end="")
    print()
```

```
$ python3 multiplication_table.py
       1    2    3    4    5    6    7    8    9   10   11   12
------------------------------------------------------------------
 1  |  1    2    3    4    5    6    7    8    9   10   11   12
 2  |  2    4    6    8   10   12   14   16   18   20   22   24
 3  |  3    6    9   12   15   18   21   24   27   30   33   36
 4  |  4    8   12   16   20   24   28   32   36   40   44   48
 5  |  5   10   15   20   25   30   35   40   45   50   55   60
 6  |  6   12   18   24   30   36   42   48   54   60   66   72
 7  |  7   14   21   28   35   42   49   56   63   70   77   84
 8  |  8   16   24   32   40   48   56   64   72   80   88   96
 9  |  9   18   27   36   45   54   63   72   81   90   99  108
10  | 10   20   30   40   50   60   70   80   90  100  110  120
11  | 11   22   33   44   55   66   77   88   99  110  121  132
12  | 12   24   36   48   60   72   84   96  108  120  132  144
$
```

Using `enumerate` provides a counter alongside an object to be iterated through.

- The `enumerate` object itself is iterable and as such can be used in a for loop.

The following example loops through a a dictionary of lists and utilizes enumerate to number the output.

numbering_output.py

```
#!/usr/bin/env python3

hardwoods = ["Oak", "Ash", "Beech", "Elm", "Mahogany", "Teak", "Walnut"]
softwoods = ["Pine", "Cedar", "Fir"]
darkwoods = ["Mahogany", "Walnut", "Ebony"]
lightwoods = ["Maple", "Ash", "Apple", "Beech"]
wood_types = {"Hard": hardwoods, "Soft": softwoods,
              "Dark": darkwoods, "Light": lightwoods}

for index, item in enumerate(wood_types.items(), start=1):
        type, collection = item  # Unpack item into type and collection
        base = index * 1000
        print(type, f"{base}'s")
        for sub_index, name in enumerate(collection, start=1):
            print(f"    {base + sub_index} {name}")
        print()
print()
```

```
$ python3 numbering_output.py
Hard 1000's
    1001 Oak
    1002 Ash
    1003 Beech
    1004 Elm
    1005 Mahogany
    1006 Teak
    1007 Walnut

Soft 2000's
    2001 Pine
    2002 Cedar
    2003 Fir

Dark 3000's
    3001 Mahogany
    3002 Walnut
    3003 Ebony

Light 4000's
    4001 Maple
    4002 Ash
    4003 Apple
    4004 Beech
$
```

The next example uses a while loop with the walrus operator to gather input from the user.

- It breaks the input into words and stores the frequency of each word in a dictionary.
- It then outputs the unique words sorted by frequency.

gathering_input.py

```
#!/usr/bin/env python3
print("Please enter some text at the prompt or <enter> to quit")
words = []
while text := input("> "):
    words.extend(text.split())

word_counts = {}
for word in words:
    word_counts[word] = word_counts.get(word, 0) + 1

unique_words = word_counts.keys()
sorted_unique_words = sorted(unique_words, key=word_counts.get,
reverse=True)

for unique_word in sorted_unique_words:
    print(f"{unique_word:>15}:{word_counts[unique_word]}")
```

```
$ python3 gathering_input.py
Please enter some text at the prompt or <enter> to quit
> Hello how are you
> I am fine, how are you
> good thanks will see you later
>
            you:3
            how:2
            are:2
          Hello:1
              I:1
             am:1
          fine,:1
           good:1
         thanks:1
           will:1
            see:1
          later:1
$
```