From the best-selling author of *Python Crash Course*

PYTHON FLASH CARDS

101 CARDS

no starch press

ERIC MATTHES

CONCEPTS AND VOCABULARY

These cards introduce a variety of important programming concepts. Understanding these concepts will help you make sense of the relevant syntax when you get to it, in Python or any other language you choose to study.

You can read these cards as a set before moving on to the syntax cards, or you might visit relevant concepts here as you work on individual topics from the syntax cards.

PROGRAMMING LANGUAGES

- What is a programming language?
- What's unique about Python?
- How does a programming language affect the way we think about solving problems?

A *programming language* is a set of rules for giving instructions to a computer. It provides the syntax for giving instructions and specifies the ways to store information, and it controls the order in which instructions are executed in a program.

Python is a *high-level programming language*, which means it takes care of many low-level tasks for you so you can focus on solving problems. For example, when you assign a value to a variable, Python deletes the variable automatically when it's no longer needed, sparing you from having to manage memory.

Every language has unique features that lead to characteristic programming styles and philosophies. Python focuses on simplicity, readability, and getting the job done.

OPERATING SYSTEMS

- What is an operating system?
- What does an operating system do?
- How does Python interact with the operating system?

The *operating system (OS)* is the software that controls the computer's inner workings.

An operating system performs low-level functionality, such as reading from and writing to memory, and interacts with hardware devices, like hard drives, RAM, CPU, graphics processors, displays, batteries, and other external devices. Windows, macOS, and Linux (such as Ubuntu and Fedora) are major operating systems.

Python is a *cross-platform programming language*. You can write Python code on any OS, and it will run on any other OS.

TERMINAL

- What is a terminal?
- How do you run a Python program from a terminal?
- How do you start a Python session from a terminal?

A *terminal* is a program that allows you to interact with the OS, and it is often referred to as the *console* or *command line*. You use a terminal (rather than going through a GUI) to issue clear, concise, text-based commands to the OS to quickly perform tasks.

You can run a Python program from a terminal using a command like this:

```
$ python hello_world.py
```

You can also start a Python session in a terminal. In a Python terminal session, each line of code executes as soon as you enter it:

```
$ python
>>> print("Hello, terminal world!")
Hello, terminal world!
```

TEXT EDITORS

- What is a text editor?
- What is syntax highlighting?
- What are some beginner-friendly text editors?
- What are some more advanced text editors?

A *text editor* is a program designed for writing and editing code.

Most text editors have features to make writing and editing code easier: *syntax highlighting*, for example, colors your code so you can quickly recognize different parts of a program—a string might be green and a method might be purple.

Sublime Text, Atom, and Geany are some commonly used beginner-friendly text editors because of their ease of use and familiar interfaces. They also have powerful features that help you work more efficiently as you learn.

Emacs and Vim are advanced text editors that were introduced in the 1970s. Their learning curve is steep, but once you learn to use them well, writing and editing code is incredibly efficient. Most Linux systems install Vim, or its predecessor vi, by default.

IDEs

- What is an IDE?
- What are some typical features of IDEs?
- What IDEs are best for Python?

An *integrated development environment (IDE)* is a text editor with powerful project management features.

Typical IDE features include debugging tools, auto-filling for certain code elements, and the ability to catch errors as you're entering code. For projects that span multiple files, the IDE looks through the files and helps maintain consistency across the project.

IDEs can make code testing easier and identify portions of your code that you could refactor. IDEs help you interact with other project elements, such as HTML and JavaScript in a web application, and help you work with a database.

Popular Python IDEs include PyCharm, PyDev, Spyder, and Visual Studio.

COMMENTS

- What are comments?
- Why are comments useful in programming?
- What kinds of comments should you write?

Comments **are lines in a program that the program ignores when it executes. They allow you to add notes about how the program works to help you and other developers understand the code.**

Use comments to explain:

- The role of important variables when you introduce them
- How you've approached a problem after considering multiple approaches
- What your functions do
- What classes are used for in the program

Writing comments will remind you what your code does when you return to it later. Comments also help teams of programmers collaborate effectively.

STYLE GUIDES

- What is a style guide?
- What kinds of recommendations are made in style guides?
- Where can you find the Python style guide?

A *style guide* offers direction on creating consistency in your code, such as how far you indent lines, your maximum line length, or how you break lengthy lines.

A style guide is not a set of rules: if you break the guidelines but follow the syntax rules, your code will still run. When you follow a style guide, your code will be consistent, and it'll be easier for you and others to focus on what it does rather than what it looks like.

Python's design ensures that programmers write more readable code with it than with other languages. Be sure to read the Python style guide, *PEP 8*, for suggestions on how to create clean and consistent code.

PROJECT SPECIFICATIONS

- What is a project specification?
- Why are specifications important?
- Who writes project specifications?

A *project specification*, or spec, lists requirements for what a program needs to do.

A clear specification is important in projects of all sizes so you have a solid idea of what to aim toward and whether your project is successful. Without a clear spec, you'll waste time and risk your project's failure.

A project specification indicates the problems that need solving and how users will interact with the program. It should specify what kinds of inputs your program will deal with, as well as the outputs the program needs to generate.

When learning to program, each exercise you attempt is a mini spec. Well-specified problems are easier to solve than poorly specified exercises. A good programmer looks for a spec when collaborating on an existing project and develops a full spec before committing to a new project.

SYNTAX

- What is syntax?
- What is a syntax error?
- What is a logical error?

The *syntax* of a programming language is the set of rules for how to write instructions in that language; syntax is like the grammar for that language. The syntax rules tell you how to store data, respond to specific conditions, and control the flow of execution in the language.

A *syntax error* occurs when a program doesn't follow the syntax rules. The language won't understand how to execute the code, even if only a minor mistake is made, such as a missing parenthesis or comma.

A *logical error* occurs when your program follows the syntax rules and runs, but doesn't do what you want it to do. You'll know you have a logical error when your project generates output but the output doesn't match the project spec. You'll need to reevaluate your approach to the problem and then modify that approach.

DEBUGGING

- What is a bug?
- What does debugging mean?
- How do you debug a program?

A *bug* is a problem in the way a program runs. *Debugging* is the process of identifying the cause of a bug and modifying code to fix the issue.

One of the first steps in debugging is to understand error messages. They tell you where the interpreter had a problem and what kind of problem occurred.

Logical errors don't result in error messages, so you need to look at the program execution to resolve them. To debug logical errors, examine the value of variables at different points in the program's execution. Do this by inserting `print()` calls into the code at strategic points, logging the values of key variables during execution, or using the debugging tools available in most IDEs. Also, write tests for your code to see which parts are working and which are not.

REFACTORING

- What is refactoring?
- When and why should you refactor code?
- What is the DRY principle?
- How do you refactor code?

***Refactoring* is the process of rewriting parts of a working program to make it simpler, more efficient, and easier to work with.**

Consider refactoring code when you've solved the same problem in multiple places. This approach follows the *don't repeat yourself*, or *DRY*, principle. Code repetition presents opportunities for errors, makes programs harder to modify, and makes programs longer and harder to read.

To refactor code, place a repeated block of code into a separate function. Then replace the repeated code with a function call. When you need to improve that code, you only need to change the code in the function for it to work throughout the program. Refactoring can help you find more efficient ways to solve problems, and separating code into different modules makes it easier to work with.

STANDARD LIBRARY

- What is a standard library?
- What is usually included in a standard library?
- What is not included in a standard library?

A *standard library* is the set of tools included in a language's standard installation. The library includes core data structures and tools for working with data in your program.

Python's standard library includes tools to help you:

- Work with a variety of data types and text
- Use mathematical functions
- Work with files and dates and times
- Make graphical user interfaces (GUIs)
- Work with networks and multimedia
- Test and debug your code and handle errors
- Distribute your programs

Seldom does a standard library include specialized data analysis tools, game frameworks, web application frameworks, and other application-specific libraries. These are available through external libraries that are updated more often than the language as a whole.

Once you understand a language's core syntax and rules, become familiar with the language's standard library tools as these can help you write code efficiently.

THIRD-PARTY LIBRARIES

- What is a third-party library?
- What is a package?
- What is a package manager?
- What kinds of third-party libraries are available for Python?

A *third-party library* adds tools and functionality that aren't covered by a language's standard library.

Third-party libraries, or *packages*, are installed through an automated *package manager*, such as pip, for consistent and secure installation. A package manager also helps keep the libraries you've installed up-to-date.

More than 100,000 packages are available through the Python package manager. For example, the `requests` package helps you write programs to access online resources, `Pillow` helps you work with images, `SQLAlchemy` makes it easier to work with databases, and `NumPy` offers tools for working with numerical data.

FRAMEWORKS

- What is a framework?
- What kinds of frameworks are available for Python?

A *framework* is a larger package that helps solve a particular problem. Some frameworks help you build games, data visualizations, and web applications.

A framework includes tools that handle common tasks within a problem area. For example, a game framework often provides a way to determine when a collision between two game elements has occurred. A web application framework usually provides a way to extract information from a database.

Simple frameworks leave it to you to make many problem-solving decisions; larger frameworks have more default approaches to common situations.

Popular Python web frameworks include Django, a large framework that provides tools to build web applications, and Flask, a bare-bones web application framework that leaves many decisions up to you. Popular game frameworks include Pygame, Kivy, and Pyglet.

ERROR HANDLING

- Why is error handling important?
- How do you handle errors in a program?
- What is an exception?

***Error handling* refers to writing code that anticipates errors that are likely to occur and then responds to those errors.**

To run successfully, a program needs the correct set of inputs and the ability to do its work, and it must return its output appropriately. Errors can happen during any of these stages. Well-written programs have error-handling code that anticipates and deals with errors that might occur at each stage.

Error handling can just be code that checks certain values and conditions before proceeding. If the values and conditions are appropriate, execution continues along one path. If not, it follows a different path.

One common approach is to check whether input is in the correct format. When a user inputs their age, you expect a positive number. You can add code to check for a positive number before continuing and then raise an exception if anything else is entered. An *exception* is a special error condition that responds to a specific kind of error.

VERSION CONTROL

- What is version control?
- What is a commit?
- What does it mean to "check out" a commit?
- What is a repository?
- Why is version control important?
- What is distributed version control?

***Version control* is the process of saving different versions of your project as it evolves. If you break a program, you can return to a previously working version, or you can simply examine how your program worked in a previous version.**

Version control systems work by allowing you to make a *commit*, which saves the current state of all the files in your project. You make a commit after implementing a new feature.

If you make a mistake and can't figure out how to fix your program, you can *check out* the previous commit and start again from a working version of your project.

A *repository* is the collection of all commits you've made to a project; it contains other resources required to manage your project as well.

A *distributed version control system* allows multiple people to make commits on the same project and provides tools for resolving conflicts in collaborative projects.

Git and Mercurial are the most common version control systems at the time of this writing.

TESTING

- What is a test?
- How do you write a test?
- What is an assertion?
- Why should you write tests for your code?
- How do tests help you add new features to a project?

A *test* is code outside your program that runs some of your program code to check whether it works correctly.

Each part of a program solves one aspect of a larger problem your program aims to solve. You can write tests that prove each part of your code behaves properly.

To write a test, you import the code you want to test, set up sample data, and run the code using that data. Then you can make assertions about the results.

An *assertion* is a statement about what a value or condition should be. For example, you might *assert* that the value of the variable age is greater than 18. If the code acts as it should, the test passes. If it doesn't, the test fails.

A good set of tests helps you find bugs before your users do. It also helps ensure that when you add new features to your projects, the code doesn't affect the behavior of existing features and your program still works.

USER INTERFACES

- What is an application programming interface?
- What is a graphical user interface?
- What is a command line interface?
- Why is each of these important?

An *application programming interface (API)* is a specification for how one program asks another program for information. A *graphical user interface (GUI)* is an easy-to-use interface that lets users click elements on the screen. A *command line interface (CLI)* allows users to interact with a program by entering commands in a terminal.

Using a GUI is the simplest way for nontechnical users to interact with a program. For example, a weather website would present information in a browser (a GUI) for general users, but it might also provide an API that technical users can work with to build projects that use real-time weather data.

APIs allow you to pull in data from external resources. Developers working on large, collaborative projects can build their own APIs to design how each program in the project communicates. Designing and building a good API requires experience and planning, but learning to *use* an API is much simpler.

A CLI enables automation and scripting in a more customizable fashion than with GUIs.

DATABASES

- What is a database?
- Why are databases important?
- What is SQL?
- How do you communicate with a database?
- What are some common databases?

A *database* is a program that allows you to store and retrieve information. Good databases are highly optimized to do this efficiently and reliably with large amounts of data.

Databases store most of the world's information. Whenever you retrieve information from a website or post to a website, that information is being read from or written to a database. Many of the projects programmers work on interact with a database.

Structured Query Language (SQL) is a language written specifically to interact with databases, but you don't need to know SQL to work with a database. Many languages and frameworks generate SQL for you, so you can work with the database through those languages. If you're serious about programming, however, it's a good idea to learn SQL.

SQLite, PostgreSQL, MySQL, SQL Server, and Oracle are the most common databases used in programming.

DATA STRUCTURES AND TYPES

- What are data structures?
- What are data types?
- Why are data structures important?
- What are the most common kinds of data structures?

Data structures **define how you store data in a programming language. Every programming language has a core set of data structures. *Data types* refer to the data structures you're using or the kind of information stored in those data structures.**

Much of programming focuses on handling data. How you represent data through code impacts what you can do with that data: if you model your data well, your program is easier to work with.

Choosing a specific approach to data modeling informs how you think about data, so knowing the available data structures in a programming language will help you best represent the information in your project.

The most common data structures include variables, sequences, mappings, classes, and objects. Python also has text-based data types, such as strings, and numerical data types, such as integers and floats.

VARIABLES

- What is a variable?
- Why are variables important?
- What is a statically typed language?
- What is a dynamically typed language?
- What makes Python a dynamic language?

In Python, a *variable* is a name attached to a piece of data. You first define a variable, then use that name when you refer to that piece of data.

In *statically typed languages*, you must declare the kind of data a variable represents when you define it. Python is a *dynamically typed language*, meaning you don't have to declare the kind of data a variable will represent. In Python, the interpreter examines the data associated with the variable throughout the life of the program and manages type issues for you.

Dynamic languages prioritize a programmer's time over efficient use of the processor, but they can still be fast and efficient when used properly. Static languages prioritize processor efficiency over a programmer's time. Programs are longer and more verbose in static languages, but can be highly optimized.

STRINGS

- What is a string?
- Why are strings important?
- What is a substring?
- What can you do with strings?

A *string* is a value made up of text. A classic example of a string is the code `"Hello, world!"`.

A string is one of the simplest data types in any language. Much of the information that's passed within and between programs is strings.

Strings are treated as a collection of characters, so the string `"123"` is different from the numerical value `123`. A *substring* is part of a string.

You can perform many actions with strings:

- Joining strings together, known as *concatenation*
- Inserting the value of a variable into a string, known as *interpolation*
- Changing a string's case
- Stripping extra whitespace from a string
- Searching for a substring within a string
- Replacing some characters in a string

In Python, strings are enclosed using single or double quotes.

NUMERICAL DATA TYPES

- What is an int?
- What is a float?
- What can you do with numerical data?
- What other kinds of numerical data types are available?

Integers and floats are the two main kinds of numerical data types. An *integer*, or *int*, is a number with no decimal part. A *float* is a number with a decimal part.

You can perform all basic arithmetic operations with numerical data and do higher-order operations, such as working with exponents, finding absolute values, and working with trigonometric functions.

Python allows you to represent complex numbers (numbers with real and imaginary parts) and work with fractions.

For computation-intensive work, third-party libraries, such as NumPy, SciPy, and pandas, can make your work easier. Many dedicated visualization libraries are available as well, such as Matplotlib, Bokeh, Pygal, and Plotly.

SEQUENCES

- What is a sequence?
- Why are sequences important?
- What kinds of things can you do with a sequence?
- What kind of sequences are available in Python?
- What is a mutable sequence?

A *sequence* is a data structure that stores a collection of items in a specific order. In some languages, a sequence can only contain items of one type; in other languages, including Python, a sequence can have items of different types.

Sequences are important when you need to store items in a specific order. For example, a list of website users might be ordered according to when each user registered.

With a sequence, you can do the following:

- Add and remove items
- Work with individual items or groups of items
- Determine whether a value is in the sequence
- Look for duplicate or unique items
- Loop through the sequence and do something with each item
- Work with items that match certain conditions.

In Python, the main sequence type is the *list*. Lists are *mutable*, meaning you can modify them after creating them. *Tuples* are immutable sequences, meaning they can't be changed after they're defined. *Strings* are a special type of sequence: each element in the sequence is a character.

MAPPINGS

- What is a mapping?
- Why are mappings important?
- What can you do with a mapping?
- What is the main mapping type in Python?

A *mapping* is a data structure that connects two or more pieces of information, commonly called a *key-value pair*. When you ask for a key, you get its associated value. One mapping structure can store many key-value pairs.

Mappings don't just store data. They allow you to establish relationships between the data. For example, you could pair each username with a password, each country with its capital, and each book with its author.

Mappings are not limited to single pieces of data: you could use a single piece of data as a key and use a sequence as the value. For example, you could pair a username with a sequence of other users to define a relationship between one user and other users. This is the core of how many social networks are implemented.

In Python, the *dictionary* is the main mapping type. Make sure you understand dictionaries, because many advanced Python data structures are represented by dictionaries.

FUNCTIONS

- What is a function?
- What are parameters and arguments?
- What is a function call?
- What are functions used for?

A *function* is a block of code that you can name and that performs a certain task. You can run the block many times by using the function name. The *definition* of a function specifies its name and the data the function needs to work. *Parameters* are the variables in the function that hold this data.

To use a function, you *call* it. When you call a function, you must provide values, or *arguments*, for each of the function's parameters.

Functions allow you to write code efficiently. When you need to perform an action more than once, wrap that code in a function and call it when you need it. When you need to change how the action is carried out, you can change the code in the function, and the improvement is applied everywhere.

CLASSES

- What is a class?
- What are attributes and methods?
- What is an instance?
- What can you do with a class?

A *class* is a structure that allows you to combine the data and functionality associated with a particular task. The variables of a class are *attributes*. The functions of a class are *methods*.

Classes can represent real-world objects or abstract ideas. After defining a class, you use it by making an *instance*, or *object*, of the class. You can make as many instances as you want from one class.

As an example, you might use a class to represent a website user. The class would have attributes associated with the user's username, password, registration date, and more. Methods would define the actions the user could take, such as registering, authenticating, logging in, and logging out. You could then make one instance for each user who registers on the site.

Many external libraries are written as classes, so learning to work with classes makes it easier to work with many existing projects.

INHERITANCE

- What is inheritance?
- What is a parent class?
- What is a child class?

***Inheritance* is the idea of basing one class on another class so the new class can use the existing methods and attributes. When your classes are modeled on similar things, use inheritance rather than starting a new class.**

Imagine you want to model a book, so you write a class called `Book`. The class attributes might be `author`, `title`, and `subject`, and methods might be `describe_book()`, `show_sample_page()`, and `review_book()`. An ebook is a kind of book, so you could make `EBook` inherit from `Book` so it has all the attributes and methods of `Book`. Then you only need to define any additional attributes and methods specific to ebooks, such as a `file_size` attribute and a `download_book()` method.

In this example, `Book` is a *parent class* and `EBook` is a *child class*. A child class inherits from the parent class. When you write a child class, you can add any attributes or methods not in the parent class, as well as customize the behaviors inherited from the parent class.

OTHER DATA TYPES

- What is a Boolean value?
- What are Boolean values used for?
- What is a null value?
- What are null values used for?

A *Boolean value* represents either true or false. In Python, these values are written as `True` and `False`.

Boolean values can represent the state of a program or a certain condition. For example, you can use variables such as `game_active`, `can_edit`, and `polling_open`, which take either a `True` or `False` value. When these values are `True`, certain code sections are enabled as the program runs.

A *null value* represents nothing. You would assign a null value to a variable that doesn't have a defined value. In Python, this is written as `None`. In logical comparisons, `None` evaluates to `False`.

The null value is useful in a many situations. For example, you can use it to make some parameters for a function optional. If a value is provided, the parameter is assigned the given value. If no value is provided, the parameter doesn't have any effect on the program. A function can also return `None` when it can't do its work, instead of generating an error that might cause the program to crash.

IF STATEMENTS

- What is an `if` statement?
- What are `if` statements used for?
- What is an `elif` statement?
- What is an `else` statement?

An `if` statement allows you to set a condition on whether certain code runs. For example, using a variable called `game_active`, you can use an `if` statement to ensure the code that starts a game runs only if this variable is set to `True`.

An `if` statement allows programmers to respond to user input, and it changes conditions during a program's execution.

You can use the `elif`, or *else if*, statement to make your program respond to multiple conditions. This code structure means "if one condition applies, do this; else if a different condition applies, do something different." You can chain together as many of these specific conditions as you need.

An `else` block runs a particular block of code when all other conditions don't apply. An `else` block must always be the last block in an `if`-`elif`-`else` chain.

LOOPS

- What is a loop?
- What are loops used for?
- What is a `for` loop?
- What is a `while` loop?
- What is a nested loop?

A *loop* is a block of code that runs multiple times. Use loops when you need to repeat actions more than once, depending on certain criteria. The two loop types are the `for` loop and the `while` loop.

A `for` loop repeats as many times as you specify or once for each item in a sequence. For example, you can write a `for` loop that runs 100 times to repeat an action 100 times. You can also write a `for` loop that runs once for each item in a collection of items.

A `while` loop runs as long as a certain condition is true. Say you have a game loop that runs as long as a `game_active` variable is set to `True`. You can then put multiple situations inside the loop that cause `game_active` to become `False` and end the game. You can also use a `while` loop to process items in a collection that might have new items added to it as the loop is running. As long as items are in the collection, you can keep working with it. When the collection is empty, the loop stops running.

A *nested* loop is a loop inside another loop. Say you need a loop that examines every pixel in an image. You could write a loop that looks at each pixel in a row and, inside that loop, add a second loop that looks at every pixel in the current column for that row.

MODULES

- What is a module?
- Why are modules important?
- How do you use a module in a program?

In Python, a *module* is a file that contains code. Modules are used to break up large programs into smaller parts for more manageable coding.

Modules compartmentalize complex programs. Programmers break up larger projects into different files, each of which focuses on one aspect of the project. This makes large projects easier to write and collaborate on in teams: each programmer can focus on only the modules they're responsible for.

To create a module, you move code that has a particular aim into a separate file. To use a module, import it into the file you're working on to make the code available. You can import the entire module or just the parts you need.

SAVING STATE

- What is a program's state?
- What is the JSON format?
- Why is JSON used to save state?

The *state* of a program is the set of values of all the variables at a given moment. *Saving state* refers to saving these values so they can be recovered if the program crashes or the user closes it.

A common way to save a program's state is to write the current values to a JSON file. *JSON (JavaScript Object Notation)* is a way of storing data that's easy for programs to read back. Originally developed for use in JavaScript programs, JSON is now used by many programming languages.

The JSON format can be read by programs and by humans. Some simple JSON data files look just like Python code. Python has libraries you can use to simplify reading and writing JSON files.

SIMPLE DATA TYPES

The simplest types of data you can store in a program are text strings and numerical data. These cards show you how to work with these simple data types.

VARIABLES

- What is a variable?
- What are the rules for naming a variable?

A *variable* is a label for a value you want to use in your program.

Variable names in Python follow two main rules:

- They must contain only letters, underscores, and numbers.
- They must start with a letter or underscore.

Good variable names are short but descriptive:

```
first_name = 'albert'
last_name = 'einstein'
username = 'einsteina'
age = 42
```

A variable can hold a small piece of information, or it can hold many gigabytes of data.

STRINGS

- What is a string?
- How do you store a string in a variable?
- How do you include tabs and newlines in a string?

A *string* is a value made up of one or more characters, surrounded by single or double quotes.

In this example, `"I love Python!"` is a string assigned to the variable `message`:

```
>>> message = "I love Python!"
>>> print(message)
I love Python!
```

Both single and double quotes work for strings, so you can use quotation marks inside a string:

```
>>> python_quote = 'I said, "I love Python!"'
>>> print(python_quote)
I said, "I love Python!"
```

Insert tabs and newlines into strings using the special sequences `\t` and `\n`:

```
>>> message = "Grocery list:\n\tmilk\n\teggs"
>>> print(message)
Grocery list:
    milk
    eggs
```

There is no limit on the length of a string.

STRING METHODS

- What is a string method?
- How do you change the case of a string?
- How do you strip whitespace from a string?

A *string method* is a function that performs an action on a string.

To change the case of a string, use the methods `title()`, `upper()`, and `lower()`:

```
>>> name = 'ella fitzgerald'
>>> name.title()
'Ella Fitzgerald'
>>> name.upper()
'ELLA FITZGERALD'
>>> name = 'Ella Fitzgerald'
>>> name.lower()
'ella fitzgerald'
```

The `lstrip()`, `rstrip()`, and `strip()` methods remove extra whitespace from strings, helpful for cleaning up data:

```
>>> name = '  jordan  '
>>> name.lstrip()
'jordan  '
>>> name.rstrip()
'  jordan'
>>> name.strip()
'jordan'
```

String methods are useful for presenting data in a certain format or cleaning up user-submitted data.

USING VARIABLES IN STRINGS

- How do you insert the value from a variable into a string?

In Python 3.6 onward, you can use a variable directly inside a string:

```
>>> username = 'efermi'
>>> print(f"Welcome back, {username}!")
Welcome back, efermi!
```

The `f` is short for *format* and tells Python to insert the value of the variable listed in braces inside the string. This tells Python to format the string using the given variable's value.

In Python 3.5 and earlier, you must use the `format()` method:

```
>>> username = 'efermi'
>>> msg = "Welcome back, {}!".format(username)
>>> print(msg)
Welcome back, efermi!
```

The placeholder `{}` will be replaced by the first variable included in the call to `format()`. You can include as many variables as you need:

```
>>> msg = "Welcome, {} and {}!".format(
      username_0, username_1)
```

COMMENTS

- How do you include a comment in your code?
- When should you include comments?

A *comment* is a line of text that's ignored by the Python interpreter. Comments allow you to leave notes for yourself or others inside a program.

Comments begin with a hash mark:

```
# Greet the user.
username = 'adrian'
print(f"Welcome back {username}!")
```

Comments are useful when writing complex code that you might need to return to. They can remind you of the reasoning behind the logic.

Clear, concise comments are a sign of a professional programmer. When considering different ways to solve a specific problem, include a comment that explains the approach you chose to take.

NUMERICAL DATA

- What are the two main types of numerical data?
- How do you find out what type of data you're working with?
- How do you convert between the two numerical types?

An *integer*, or *int*, is a number that doesn't have a decimal point. A *float* is a number that does.

The `type()` function identifies the data type of its argument.

```
>>> type(3)
<class 'int'>
>>> type(3.5)
<class 'float'>
```

The `float()` function converts an integer to a float:

```
>>> float(3)
3.0
```

Sometimes numerical information starts as a string. The `float()` function converts properly formatted strings to floats:

```
>>> float('3.5')
3.5
```

The `int()` function converts floats to integers by dropping the number's decimal portion, not by rounding:

```
>>> int(3.0)
3
>>> int(3.9)
3
```

The `int()` function also converts strings to integers:

```
>>> int('3')
3
```

NUMERICAL OPERATIONS

- How do you represent basic mathematical operations?
- How does the operation you're using affect the type of your output?
- How do you force Python to use a nonstandard order of operations?

Using addition, subtraction, or multiplication with integers returns an integer:

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
```

Using addition, subtraction, or multiplication with at least one float returns a float. All division operations return floats:

```
>>> 2.0 + 3
5.0
>>> 3.0 - 2
1.0
>>> 2.0 * 3
6.0
>>> 10 / 5
2.0
```

Two asterisks (**) represent an exponent. To raise 2 to the power of 3, enter:

```
>>> 2 ** 3
8
```

Use parentheses to force a nonstandard order of operations:

```
>>> (2 + 3) * 4
20
```

WORKING WITH NUMERICAL DATA

- How do you round numbers?
- How do you get the absolute value of a number?
- How do you convert a base-10 number to binary, octal, or hexadecimal?
- How do you represent complex numbers?

The `round()` function rounds a float to the given number of decimal places. Passing `round()` a negative argument results in multiples of 10:

```
>>> round(3.1415926, 2)
3.14
>>> round(1234, -2)
1200
```

The `abs()` function returns the absolute value of a number:

```
>>> abs(-5)
5
```

The `bin()`, `oct()`, and `hex()` functions convert base-10 numbers to base-2, base-8, and base-16 numbers:

```
>>> bin(20)
'0b10100'
>>> oct(20)
'0o24'
>>> hex(20)
'0x14'
```

The first two characters of the output indicate the new base.

The `complex()` function represents complex numbers:

```
>>> complex(2, 3)
(2+3j)
```

USING THE MATH LIBRARY

- How do you take the square root of a number?
- How do you use a mathematical constant such as pi?
- How do you use trigonometric and logarithmic functions?

The `math` library provides a number of functions for operations like taking square roots, using trigonometric functions, and working with mathematical constants:

```
>>> import math
>>> math.sqrt(9)
3.0
>>> PI = math.pi
>>> PI
3.141592653589793
>>> math.sin(PI/2)
1.0
>>> math.log(100, 10)
2.0
```

The standard library and third-party libraries also offer more mathematical functions. Before writing your own mathematical algorithm to solve a problem, be sure to look for an existing function or library.

LISTS AND TUPLES

Lists allow you to store multiple pieces of information—as many as you want—in one place. Once you've defined a list, you can access and work with individual items, a section of the list, or all the information in the list at once.

Understanding how lists work gives you a solid foundation for working with more complex data types. These cards cover lists and tuples, which are similar to lists.

LISTS

- What is a list?
- How do you define a list?
- How do you add items to a list?

A *list* is a collection of items in a specific order. Square brackets indicate a list.

You access list items through their position in the list, or *index*. The first item is always index zero; a negative index accesses the list from the other end:

```
>>> vowels = ['e', 'i', 'o']
>>> vowels[0]
'e'
>>> vowels[-1]
'o'
```

Use `append()` to add single items to the end of a list:

```
>>> vowels.append('u')
>>> vowels
['e', 'i', 'o', 'u']
```

Use `insert()` to insert items at any position using the index of that position:

```
>>> vowels.insert(0, 'a')
>>> vowels
['a', 'e', 'i', 'o', 'u']
```

Modify an element in a list using its index:

```
>>> vowels[-1] = 'y'
>>> vowels
['a', 'e', 'i', 'o', 'y']
```

This replaces the last item with `y`.

REMOVING ITEMS FROM LISTS

- How do you remove an item from a list?
- How do you remove an item at a certain position from a list?

The `remove()` method removes a specific item from a list:

```
>>> airports = ['ANC', 'JNU', 'SEA', 'SIT']
>>> airports.remove('SEA')
>>> airports
['ANC', 'JNU', 'SIT']
```

If an item appears more than once in the list, only its first appearance is removed.

The `del` keyword deletes an item at a specific position in the list using its index:

```
>>> airports = ['ANC', 'JNU', 'SEA', 'SIT']
>>> del airports[2]
>>> airports
['ANC', 'JNU', 'SIT']
```

The `pop()` method removes and returns the last item in a list. The `pop()` method can also accept an index and will remove and return the item at that index:

```
>>> airports = ['ANC', 'JNU', 'SEA', 'SIT']
>>> airport = airports.pop()
>>> airport
'SIT'
>>> airport = airports.pop(0)
>>> airport
'ANC'
```

SLICING A LIST

- What is a slice?
- How do you make a slice from the beginning of a list?
- How do you make a slice at the end of a list?

A *slice* is part of a list that begins with the item at the first index specified and stops with the item before the last index specified:

```
>>> states = ['CA', 'CO', 'CT', 'DE', 'FL', 'GA']
>>> states[2:4]
['CT', 'DE']
```

Omit the first index, and the slice starts with the first item in the list:

```
>>> states[:2]
['CA', 'CO']
```

Omit the second index, and the slice goes to the end of the list:

```
>>> states[3:]
['DE', 'FL', 'GA']
```

You can construct a slice that returns any part of a list:

First two items	`states[:2]`
Last two items	`states[-2:]`
Range of items	`states[2:4]`

COPYING A LIST

- What can you do with a copy of a list?
- How do you make a copy of a list?

A copy of a list lets you work with the contents of the copied list without affecting the original list.

Make a copy of a list by slicing while omitting both indexes. This generates a slice from the start to the end of the list.

```
>>> states = ['CA', 'CO', 'CT', 'DE']
>>> my_states = states[:]
>>> my_states
['CA', 'CO', 'CT', 'DE']
```

Changes you make to the new list don't affect the original list:

```
>>> my_states.append('HI')
>>> my_states
['CA', 'CO', 'CT', 'DE', 'HI']
>>> states
['CA', 'CO', 'CT', 'DE']
```

LOOPING THROUGH LISTS

- What is a `for` loop?
- How do you loop through a list?
- How do you loop through a section of a list?

A for loop lets you work with each item in a list, one item at a time.

Use a `for` loop to loop through all items in a list:

```
countries = ['CAN', 'CHL', 'IND', 'AUS']
for country in countries:
    print(f"I'm flying to {country}.")
```

```
I'm flying to CAN.
I'm flying to CHL.
I'm flying to IND.
I'm flying to AUS.
```

Loop through part of a list using a slice:

```
for country in countries[:2]:
    print(f"I've been to {country}.")
```

```
I've been to CAN.
I've been to CHL.
```

SORTING LISTS

- How do you sort a list temporarily?
- How do you sort a list permanently?

The `sorted()` function returns a copy of a list in a natural sorted order. The order of the original list is not affected:

```
>>> vowels = ['a', 'i', 'e', 'u', 'o']
>>> sorted(vowels)
['a', 'e', 'i', 'o', 'u']
>>> vowels
['a', 'i', 'e', 'u', 'o']
```

To sort a list, use the `sort()` method:

```
>>> vowels.sort()
>>> vowels
['a', 'e', 'i', 'o', 'u']
```

To put a numerical list in order, use any sorting function:

```
>>> nums = [1, 3, 2, 5]
>>> sorted(nums)
[1, 2, 3, 5]
```

REVERSE SORTING LISTS

- How do you sort a list in reverse alphabetical order?
- How do you reverse a list's order?

To sort a list in reverse natural order, use the `reverse` argument set to `True`:

```
>>> vowels = ['a', 'i', 'e', 'u', 'o']
>>> sorted(vowels, reverse=True)
['u', 'o', 'i', 'e', 'a']
```

The `reverse` argument works with `sort()` and `sorted()`.

To reverse the original order of a list, use the `reverse()` method:

```
>>> vowels = ['a', 'i', 'e', 'u', 'o']
>>> vowels.reverse()
>>> vowels
['o', 'u', 'e', 'i', 'a']
```

The `reverse` argument to `sort()` or `sorted()` results in reverse natural order; the `reverse()` method reverses the original order of the list.

NUMERICAL LISTS

- How do you make a simple list of numbers?
- How do you make a list of numbers that follows a specific pattern?

The `range()` function generates a series of numbers. Giving `range()` one argument returns a series of numbers from 0 up to, not including, the number:

```
>>> nums = list(range(5))
>>> nums
[0, 1, 2, 3, 4]
```

Pass two values to create a range that starts with the first value and ends at one less than the second value:

```
>>> high_nums = list(range(95, 100))
>>> high_nums
[95, 96, 97, 98, 99]
```

Use a third argument to specify a distance to skip between numbers in a list:

```
>>> odds = list(range(1, 10, 2))
>>> odds
[1, 3, 5, 7, 9]
```

To define a pattern, make an empty list and then modify each number in a range. This lists the powers of 10:

```
nums = []
for exp in range(5):
    nums.append(10**exp)
print(nums)
```

```
[1, 10, 100, 1000, 10000]
```

LIST COMPREHENSIONS

- What is a list comprehension?
- How do you use a list comprehension?

A *list comprehension* is a compact way of defining a list in one line:

```
>>> nums = [10**exp for exp in range(5)]
>>> nums
[1, 10, 100, 1000, 10000]
```

A list comprehension uses square brackets, an expression that generates each item in the list, and a `for` loop as the basis for the list.

A list comprehension can modify each item in an existing list:

```
>>> states = ['CA', 'CO', 'CT']
>>> lower_states = [state.lower()
...   for state in states]
>>> lower_states
['ca', 'co', 'ct']
```

TUPLES

- What is a tuple?
- How do you define a tuple?
- How do you work with a tuple?

A *tuple* is an ordered collection of items that can't be modified. It is usually indicated by parentheses:

```
>>> elementary_grades = (2, 3, 4)
```

Elements in a tuple are accessed using indexes:

```
>>> elementary_grades[0]
2
>>> elementary_grades[-1]
4
>>> elementary_grades[:2]
(2, 3)
```

You can loop through a tuple using a `for` loop:

```
elementary_grades = (2, 3, 4)
for grade in elementary_grades:
    print(f"Welcome to grade {grade}!")
```

```
Welcome to grade 2!
Welcome to grade 3!
Welcome to grade 4!
```

If you try to modify an element in a tuple, you'll get an error. Tuples are useful when you have a collection of items that shouldn't change throughout the life of your program.

DICTIONARIES

Dictionaries allow you to establish connections between individual pieces of information, such as a state's abbreviation and its full name. There is no limit to the amount of information you can store in a dictionary.

As with lists, you can work with all the information in a dictionary or with any part of a dictionary. These cards show you how to create and work with dictionaries.

ABOUT DICTIONARIES

- What is a dictionary?
- How do you define a dictionary?
- How do you get information out of a dictionary?

A *dictionary* is a set of items; each item consists of a *key* and a *value*. You define a dictionary using braces:

```
>>> capitals = {
...     'AK': 'Juneau',
...     'AL': 'Montgomery',
...     'AZ': 'Phoenix',
... }
```

You access the value associated with a key by stating the dictionary name, followed by the key in brackets:

```
>>> capitals['AK']
'Juneau'
```

Add items to an existing dictionary by placing the new key in brackets and providing the value:

```
>>> capitals['AR'] = 'Little Rock'
```

Remove items from a dictionary using the `del` keyword:

```
>>> del capitals['AR']
```

This removes both the key and its associated value from the dictionary.

DICTIONARY METHODS

- What does the dictionary method `get()` do?
- What does the method `keys()` do?
- What does the method `values()` do?
- What does the method `items()` do?

The `get()` method returns the value associated with a key if that key exists in the dictionary. If the key doesn't exist, `get()` returns `None`, not an error.

You can also pass a default value to be returned if the key doesn't exist (example continues from Card 4.1):

```
>>> capitals.get('AK')
'Juneau'
>>> capitals.get('WY')
>>> capitals.get('WY', 'unknown')
'unknown'
```

The methods `keys()`, `values()`, and `items()` return different aspects of a dictionary that also help you loop through the dictionary:

```
>>> capitals.keys()
dict_keys(['AK', 'AL', 'AZ'])
>>> capitals.values()
dict_values(['Juneau', 'Montgomery', 'Phoenix'])
>>> capitals.items()
dict_items([('AK', 'Juneau'),
('AL', 'Montgomery'), ('AZ', 'Phoenix')])
```

Choose one of these methods based on what you're doing with each item.

LOOPING THROUGH A DICTIONARY

- How do you loop through all the keys in a dictionary?
- How do you loop through all the values in a dictionary?
- How do you loop through all the key-value pairs in a dictionary?

Loop through a dictionary to access just the keys. This is the default behavior when looping through a dictionary (example continues from Card 4.1):

```
for state in capitals:
    print(f"State: {state}")
```

```
State: AK
State: AL
State: AZ
```

You can also access just the values in a dictionary:

```
for capital in capitals.values():
    print(f"Capital: {capital}")
```

```
Capital: Juneau
Capital: Montgomery
Capital: Phoenix
```

To work with both keys and values, use the `items()` method:

```
for state, capital in capitals.items():
    print(f"{capital}, {state}")
```

```
Juneau, AK
Montgomery, AL
Phoenix, AZ
```

EXAMPLE DICTIONARIES

- What does a dictionary of multiple similar objects look like?
- What does a dictionary that represents only one object look like?

One use of a dictionary is to represent a collection of key-value pairs with a consistent structure, such as a glossary:

```
python_terms = {
    'loop': 'repeated action',
    'list': 'ordered collection',
    'dictionary': 'keys and values',
}
```

In this example, every key represents a term, and every value represents a meaning.

Another use is to represent information about one kind of object or topic. This dictionary stores information about a musician:

```
c_berry = {
    'first': 'Chuck',
    'last': 'Berry',
    'birth': '10/18/1926',
    'death': '3/18/2017',
    'style': 'rock',
}
```

NESTING: A LIST OF DICTIONARIES

- How do you store a set of dictionaries in a list?
- How do you work with dictionaries stored in a list?

If a dictionary contains a variety of information about an object, it can be helpful to store a collection of dictionaries in a list:

```
musicians = []

e_fitzgerald = {
    'first': 'Ella'
    --snip--
}
musicians.append(ella)

j_joplin = {
    'first': 'Janis',
    --snip--
}
musicians.append(janis)
--snip--
```

To work with this kind of data, loop through the list, working with each dictionary in turn:

```
for musician in musicians:
    full_name = f"{musician['first']} "
    full_name += f"{musician['last']}"
    print(full_name)
```

```
Ella Fitzgerald
Janis Joplin
```

NESTING: A LIST IN A DICTIONARY

- How do you use a list inside a dictionary?
- How do you work with a list that's stored in a dictionary?

A dictionary key can be associated with a list of values:

```
states_visited = {
    'Eric': ['AK', 'WY', 'WA', 'NH'],
    'Isaac': ['CO', 'NY', 'AK'],
    --snip--
}
```

When you loop through the items, each value will be a list. To do this, nest a `for` loop for the list inside a `for` loop for the dictionary:

```
for name, states in states_visited.items():
    print(f"{name} has visited:")
    for state in states:
        print(f"  {state}")
```

```
Eric has visited:
  AK
  WY
  WA
  NH
Isaac has visited:
  CO
  NY
  AK
```

CONDITIONAL STATEMENTS

Conditional statements control the flow of your program based on conditions you set. This allows your code to respond in a certain way depending on those conditions. You can define any set of conditions and write code that responds exactly the way you need it to.

ABOUT CONDITIONAL STATEMENTS

- What is a conditional statement?
- What is a Boolean value?
- What kinds of logical tests can you perform in a conditional statement?

A *conditional statement* checks for a certain true condition before executing its code:

```
if truck_cost > 10000:
    print("That's too much!")
```

Every conditional statement has a *logical test* that evaluates a condition to determine whether it's `True` or `False`. These values are *Boolean values*.

Here are some commonly used logical tests:

`age == 21`	Equality
`age != 21`	Inequality
`age < 21`	Less than
`age <= 21`	Less than or equal to
`age > 21`	Greater than
`age >= 21`	Greater than or equal to
`name == 'jesse'`	String equality
`name != 'jesse'`	String inequality

You can also check whether or not an item is in a list:

```
my_state in northwest_states
my_state not in northwest_states
```

IF STATEMENTS

- How do you write a simple `if` statement?

An `if` statement consists of a conditional test on one line and a statement or block of statements that run if the test returns `True`:

```
if plant_height < 3:
    print("Keep it in the greenhouse!")
```

An `if` statement can also test a Boolean value and run if that value is `True`:

```
if game_active:
    print("Let's play!")
```

IF-ELSE STATEMENTS

- How do you write an `if-else` statement?
- How do you write a series of `if-elif-else` statements?

An `if-else` statement has an `if` statement that runs if the condition is `True` and an `else` clause that runs if the condition is `False`:

```
if plant_height < 3:
    print("Keep it in the greenhouse!")
else:
    print("Move it outside.")
```

An `if-elif-else` block has an `if` statement, a series of test conditions if the first test fails, and an `else` block that runs if all tests fail:

```
if plant_height < 3:
    print("Keep it in the greenhouse!")
elif plant_height < 15:
    print("Move it outside.")
else:
    print("Ready to harvest!")
```

An `if-elif` block doesn't require an `else` block.

USER INPUT

- How do you prompt the user for input?
- How do you work with the data entered by a user?

The `input()` function pauses a program and waits for the user to enter data:

```
name = input("What's your name? ")
print(f"Hello, {name}!")
```

```
What's your name? Cedar
Hello, Cedar!
```

All data entered is converted to a string. Numerical data needs to be converted to the appropriate type before you can work with it:

```
price = input("How much for the truck? ")
price = float(price)
if price < 10000:
    print("I'll take it.")
```

```
How much for the truck? 7500.00
I'll take it.
```

WHILE LOOPS

- What is a `while` loop?
- How do you use a list with a `while` loop?

A while loop runs as long as a condition is True:

```
num = 0
while num < 3:
    print(num)
    num += 1
```

```
0
1
2
```

A list with at least one item evaluates to `True`. This loop runs until the list is empty:

```
bugs = ['bug 1', 'bug 2', 'bug 3']
while bugs:
    bug = bugs.pop(0)
    print(f"Fixing {bug}.")
print("All bugs fixed.")
```

```
Fixing bug 1.
Fixing bug 2.
Fixing bug 3.
All bugs fixed.
```

THE BREAK AND CONTINUE STATEMENTS

- What does `break` do to a loop?
- What does `continue` do to a loop?

The `break` statement ends a loop when a certain condition occurs:

```
while True:
    name = input("What's your name? ")
    if name == 'quit':
        break
    print(f"Hello, {name}!")
```

```
What's your name? Cedar
Hello, Cedar!
What's your name? quit
```

The `continue` keyword breaks the loop and returns to the beginning of the loop:

```
while True:
    age = input("How old are you? ")
    age = int(age)
    if age < 0:
        print("That makes no sense!")
        continue
    print(f"{age} is a great age!")
```

```
How old are you? 17
17 is a great age!
How old are you? -5
That makes no sense!
How old are you? 5
5 is a great age!
```

FUNCTIONS

Functions allow you to define a set of actions and then repeat that set of actions at any time by calling the function name in your code. When creating a function, you can specify exactly the information it needs to do its work, and your function can return exactly the values you need it to.

Using functions will help you write code that is well organized, concise, and easy to build upon.

ABOUT FUNCTIONS

- What is a function?
- How do you define functions?
- How do you call functions?

A *function* is a block of code you give a name to so you can use it again. Calling its name runs the entire block of code.

This one-line function displays a greeting:

```
def hello():
    """Greet everyone."""
    print("Hello, everyone!")

hello()
```

To create a function, *define* it with the keyword `def`, followed by the function's name, a set of parentheses, and a colon at the end.

The next line is a *docstring*: a short description of the function in triple quotes. The function's *body* contains the Python code that runs when you call the function. The body has no line limit and is indented under the function definition.

To *call* a function, use the function's name followed by a set of parentheses. Here's the output from calling our `hello()` function:

```
Hello, everyone!
```

PASSING ARGUMENTS

- What is a parameter?
- What is an argument?
- How do you write a function that takes in information?
- How do you pass arguments to functions?

A *parameter* is information the function needs to do its work. An *argument* is a value passed to a function.

To define parameters for a function, place them within the parentheses in the function's definition. This creates a `hello()` function that requires `name` data:

```
def hello(name):
    """Greet someone by name."""
    print(f"Hello, {name}!")
```

To run, this function needs a name argument, which it stores in the `name` variable and uses when it calls `print()`.

Pass an argument to a function by including a value in the parentheses when you call the function:

```
hello("Gretchen")
```

The argument provides a value for the parameter `name`, allowing the `hello()` function to generate a personalized greeting:

```
Hello, Gretchen!
```

Sometimes *parameter* and *argument* are used interchangeably.

POSITIONAL ARGUMENTS

- What are positional arguments?
- How do you use positional arguments in a function call?
- What happens if you use positional arguments in the wrong order?

When a function needs more than one value, it must match the values it receives to its parameters. Using *positional* arguments, the function matches the first value it receives to the first parameter, and so on.

This function accepts a message and a username and posts a message summary:

```
def post_msg(msg, user):
    """Post a message from a user."""
    print(f'{user}: {message}')

post_msg("I love the internet!", "Willie")
post_msg("Me too!", "Ever")
```

```
Willie: I love the internet!
Ever: Me too!
```

The order of the arguments must match the order of the parameters in the function definition, or the output won't make sense:

```
post_msg("Willie", "I love the internet!")
```

This swaps the username and message values:

```
I love the internet!: Willie
```

ARBITRARY POSITIONAL ARGUMENTS

- How do you write a function that accepts an unknown number of arguments?
- How do you call a function that accepts an arbitrary number of arguments?

An asterisk (*) before a parameter name allows a function to receive any number of positional arguments. Use it when you don't know how many arguments the function will receive.

This function accepts a username and the number of messages the user posts. The arguments make up a tuple:

```
def post_messages(user, *msgs):
    """Post all of a user's messages."""
    print(f"{user} said:")
    for msg in msgs:
        print(f"  {msg}")

post_messages('Ahna', "I love mountains.",
        "I love rivers.",
        "I love the ocean.")
```

The first argument must provide a value for the first parameter. The rest are placed into the tuple `msgs`. Here's the output:

```
Ahna said:
  I love mountains.
  I love rivers.
  I love the ocean.
```

A function can only have one parameter that collects an arbitrary number of positional arguments.

KEYWORD ARGUMENTS

- What are keyword arguments?
- How do you use keyword arguments in a function call?

***Keyword arguments* specify the parameter a value will be assigned to so you can pass arguments in any order.**

In the function call, include the name of the parameter and the value to assign to that parameter:

```
def post_msg(msg, user):
    """Post a message from a user."""
    print(f'{user}: {message}')

post_msg(user="Willie",
         msg="I love the internet!")
```

Python assigns the value `"Willie"` to the parameter `user` and the value `"I love the internet!"` to the parameter `msg`:

```
Willie: I love the internet!
```

The order of keyword arguments in a function call doesn't matter.

ARBITRARY KEYWORD ARGUMENTS

- How do you write a function that accepts an arbitrary number of keyword arguments?
- How do you call a function with an arbitrary number of keyword arguments?

Placing a double asterisk (**) before a parameter allows a function to accept an arbitrary number of keyword arguments. Use them when you don't know the kind of information the function will receive.

A parameter with ** collects any remaining name-value pairs from the function call and adds them to a dictionary. This function accepts an arbitrary amount of user information:

```
def desc_user(user, **desc):
    """Describe a user."""
    print(f"Description of {user}:")
    for key, value in desc.items():
        print(f"  {key}: {value}")

desc_user('anton', active=True,
        email='anton@example.com',
        num_posts=578)
```

The first argument must be the user. Then any remaining keyword arguments are packed into the `desc` dictionary:

```
Description of anton:
  active: True
  email: anton@example.com
  num_posts: 578
```

Each function can only have one parameter that collects an arbitrary number of keyword arguments.

DEFAULT VALUES

- How do you assign a default value for a parameter?
- How do you call a function that has a default value for a parameter?

Setting a parameter's *default value* allows the function call to use the default value unless another value is specified with the call.

This function serves a cup of coffee. You can specify the cup size, but if you don't, you'll get a 12 oz cup:

```
def coffee(size=12):
    """Serve a cup of coffee."""
    print(f"Here's your {size} oz coffee!")

coffee()
coffee(16)
```

Here we get a 12 oz cup and a 16 oz cup:

```
Here's your 12 oz coffee!
Here's your 16 oz coffee!
```

Default values are helpful because they make some arguments optional.

RETURN VALUES

- What is a return value?
- How do you write a function that returns a value?

A function sends a *return value* back to the line that called the function, so the value can be used later in the code. The calling line uses a variable to store the return value.

This function accepts a rectangle's width and height and returns the area:

```
def area_rect(w, h):
    """Calculate the area of a rectangle."""
    area = w * h
    return area

my_area = area_rect(5, 4)
print(my_area)
```

When the function is called, the return value is stored in the variable `my_area`. Here's the output:

```
20
```

You can return any kind of value. To return more than one value, place the values in parentheses to return them as a tuple.

MODULES

- What is a module?
- How do you create a module?

A *module* is a Python file that contains multiple functions or classes. By storing functions in this separate file, you can call them in programs without having to enter the full function code each time.

This *area_functions.py* module has two functions:

```
"""A set of functions for calculating areas."""

def area_rect(w, h):
    """Calculate the area of a rectangle."""
    --snip--

def area_circle(r):
    """Calculate the area of a circle."""
    --snip--
```

No function *calls* are in this file, so this code doesn't run until the main code calls it. When you add to and improve functions in modules, the changes are immediately available to all programs that use them.

IMPORTING FUNCTIONS

- How do you import an entire module and use each of the functions in the module?
- How do you give a module an alias when importing it?

Use the `import` keyword to import a module and access its functions.

After importing, use any module function by providing the module's name, a dot, and the function's name:

```
import area_functions

my_area = area_functions.area_rect(4, 5)
print(my_area)
```

This code has the same output as if it contained all the function code:

```
20
```

Use `as` to give a module an alias and use shorter function calls. Here we use `af` for `area_functions`:

```
import area_functions as af

my_area = af.area_rect(4, 5)
print(my_area)
```

IMPORTING SPECIFIC FUNCTIONS

- How do you import one function from a module?
- How do you import several individual functions from a module?

Use `from` with `import` to import a specific function from a module rather than the entire module. Then use the function's name as if it were defined within the program.

Here we import from the `area_functions` module from Card 6.9:

```
from area_functions import area_rect

my_area = area_rect(4, 5)
print(my_area)
```

To import multiple functions from one module, separate them with commas:

```
from area_functions import area_rect, area_circle
```

Use an asterisk to import all the functions in a module at once:

```
from area_functions import *
```

This last approach is not recommended because it can cause conflicts. A module function might override a program function if they have the same name. If you need all the module functions, import the entire module and use an alias.

CLASSES

Classes are at the heart of object-oriented programming. They allow you to store both information and actions to perform on that information in one place. Understanding classes will allow you to write code that models just about anything in the real world.

ABOUT CLASSES

- What is a class?
- What is an attribute?
- What is a method?
- How do you define a class?

A *class* organizes code that combines data and actions. To use a class, you make an *instance* of the class. An *attribute* is a variable associated with a class, and a *method* is a function associated with a class.

This class represents a trail, such as a hiking trail, with two attributes, `self.dest` and `self.len`, and one method, `__init__()`:

```
class Trail():
    """A class to represent trails."""

    def __init__(self, dest, len=0):
        self.dest = dest
        self.len = len
```

Python classes generally use initial capital letters. When an instance of a class is made, the `__init__()` method runs automatically. In this code, `__init__()` requires one argument, `dest`, and has one optional parameter, `len`.

The first parameter, `self`, references the current instance of the class. Attaching the values `dest` and `len` to `self` makes them available to all methods in the class and any instance of the class.

METHODS

- How do you define a method?
- How do you make an instance of a class?

After defining a class, you can add methods that allow you to take actions using the class.

This method generates a description of a trail and prints that description:

```
class Trail():
    """A class to represent trails."""

    def __init__(self, dest, len=0):
        self.dest = dest
        self.len = len

    def describe_trail(self):
        """Print a description of trail."""
        desc = f"This trail goes to {self.dest}."
        if self.len:
            desc += f"\nThe trail is {self.len}km."
        print(desc)
```

Each method is automatically sent a reference to the instance of the class, so each method needs a `self` parameter. Then the method can use any attribute from the class. In this code, the description includes the trail's destination and its length if it's specified.

MAKING INSTANCES

- How do you make an instance of a class?
- How do you access the value of an attribute through an instance?
- How do you use an instance to call methods?

To use class attributes and methods, you must make instances, or *objects*, from the class. The class defines a set of values and actions that represents any trail. An *instance* is a specific set of values that represents a specific trail.

Here we make an instance of `Trail` and call the method `describe_trail()`:

```
verst = Trail("Mt. Verstovia", 4)
print(f"Destination: {verst.dest}")
verst.describe_trail()
```

You provide the class name and include any arguments the `__init__()` method requires. You assign the instance to a variable, and then you can access attributes and call methods using dot notation.

Here's the output:

```
Destination: Mt. Verstovia
This trail goes to Mt. Verstovia.
The trail is 4km.
```

ADDING METHODS

- How do you define a class with multiple methods?

A class can have as many methods as it needs.

Here's `Trail` with a second method defined:

```
class Trail():
    """A class to represent trails."""
        --snip--

    def run_trail(self):
        """Simulate running the trail."""
        print(f"Running to {self.dest}.")
```

Each method has the `self` parameter, giving it access to all the attributes in the class.

Here we make an instance representing a specific trail and call the two methods:

```
verst = Trail("Mt. Verstovia", 4)
verst.describe_trail()
verst.run_trail()
```

Here's the output:

```
This trail goes to Mt. Verstovia.
The trail is 4km.
Running to Mt. Verstovia.
```

MULTIPLE INSTANCES

- How do you make multiple instances from a class?

You can make as many instances as you need from a class.

Here, the same class generates two trails:

```
verst = Trail("Mt. Verstovia", 4)
verst.describe_trail()
verst.run_trail()

ms = Trail("Middle Sister", 10)
ms.describe_trail()
ms.run_trail()
```

Here's the output:

```
This trail goes to Mt. Verstovia.
The trail is 4km.
Running to Mt. Verstovia.
This trail goes to Middle Sister.
The trail is 10km.
Running to Middle Sister.
```

Each instance maintains its own set of attributes.

INHERITANCE

- What is inheritance?
- What is a parent class?
- What is a child class?
- How do you use inheritance?

By using *inheritance*, you can create a new class that inherits the attributes and methods of the original class.

We'll create a class to represent a bike trail: the *parent class* is `Trail`, and the *child class* is `BikeTrail`.

You define a child class like a normal class but include the parent class name in parentheses:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        super().__init__(dest, len)
        self.paved = True
        self.bikes_only = True
```

The child class `__init__()` method often needs to call the parent class `__init__()` method by using `super()`. The `super()` function references the parent class and allows you to call methods from the parent class.

The child class can define additional attributes particular to itself. For example the `self.bikes_only` attribute indicates whether the trail is closed to nonbicyclists.

CHILD CLASS METHODS

- How do you write a new method for a child class?

A child class can define its own methods, giving it specific behavior the parent class lacks.

The new `BikeTrail` method, `ride_trail()`, prints a message that's only appropriate for a bike trail:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        --snip--

    def ride_trail(self):
        """Simulate riding the trail."""
        print(f"Riding to {self.dest}.")

cross_trail = BikeTrail("Harbor Mountain", 5)
cross_trail.ride_trail()
```

Here's the output:

```
Riding to Harbor Mountain.
```

OVERRIDING PARENT CLASS METHODS

- How does a child class modify the behavior of a method from the parent class?

A child class can customize the behavior of parent class methods by using `super()` to access those methods.

This `BikeTrail` class modifies the behavior of the method `run_trail()`:

```
class BikeTrail(Trail):
    """Represent a bike trail."""

    def __init__(self, dest, len=0):
        --snip--

    def run_trail(self):
        """Simulate running the trail."""
        if self.bikes_only:
            print("You can't run this trail!")
        else:
            super().run_trail()

cross_trail = BikeTrail("Harbor Mountain", 5)
cross_trail.run_trail()
```

The method first checks if this is a bikes-only trail; if it is, an appropriate message prints:

```
You can't run this trail!
```

If the trail allows runners, the method calls the parent method `run_trail()`, inheriting the behavior of a regular `Trail`.

STORING CLASSES IN MODULES

- How do you store a class in a module?
- How do you import a class from a module?
- How do you use an imported class?

You store classes in modules and import them the same way as functions.

Here we store the classes `Trail` and `BikeTrail` in the *trails.py* file:

```
class Trail():
    """A class to represent trails."""
    --snip--

class BikeTrail(Trail):
    """Represent a bike trail."""
    --snip--
```

Now we can import these classes into another file:

```
from trails import Trail, BikeTrail

verst = Trail("Mt. Verstovia", 4)
cross_trail = BikeTrail("Harbor Mt", 5)
```

Use dot notation to import the entire module:

```
import trails

verst = trails.Trail("Mt. Verstovia", 4)
cross_trail = trails.BikeTrail("Harbor Mt", 5)
```

Use aliases when importing classes. Store classes to organize your code and make the classes available to multiple programs.

TESTING

Once your project is doing what you want it to do, you'll want a way to check whether your code will continue to work, even as you modify what you've written and add new features.

Writing and running tests for your code will tell you whether it's performing as you intended. These cards cover some common tools for testing code.

THE UNITTEST MODULE

- What is the `unittest` module?
- How do you write a test case?
- What does the `unittest.assertEqual()` method do?
- What other assert methods are available?

The `unittest` module provides tools for code testing, such as the `TestCase` class. This class's assertion methods ensure your code generates proper results; its `setUp()` method allows you to use the same initial conditions in multiple tests.

To test a function or class, import the test code into a new file and write a test case that inherits from `unittest.TestCase`. Then write methods to test code behavior. Use the module when modifying code, fixing bugs, and adding features.

You can make these assertions in your testing code:

`assertEqual()`	Two values match
`assertNotEqual()`	Two values don't match
`assertTrue()`	A value is True
`assertFalse()`	A value is False
`assertIn()`	An item is in a list
`assertnotIn()`	An item is not in a list

TESTING FUNCTIONS

- How do you write a test for a function?

Here's a test for the `area_rect()` function from Card 6.8:

```
import unittest
import area_functions as af

class AFTestCase(unittest.TestCase):
    """Tests for area_functions.py."""

    def test_rect_area(self):
        """Test a simple rectangle."""
        area = af.area_rect(3, 4)
        self.assertEqual(area, 12)

if __name__ == '__main__':
    unittest.main()
```

You import `unittest` and the module you're testing. Test class names usually end in `TestCase`; a prefix indicates what they test. The test class must be a child class of `unittest.TestCase`.

Each test is a method. Every method that starts with `test_` runs when the tests run. The method `test_area_rect()` calls `area_rect()` with the rectangle dimensions and then tests for the correct area.

RUNNING A PASSING TEST

- How do you run a test for a function?
- What does a passing test look like?

To run a test for a function, run the file that has the function's tests:

```
$ python test_area_functions.py
.
------------------------------
Ran 1 test in 0.000s
OK
```

Python prints a dot for a passing test. The results for running only one test are not all that helpful, but long test case output is much more useful. A failing test shows an `E` or an `F` for an *error* or a *failed assertion*.

When complete, the test case runs a summary indicating the number of tests and their runtime. The `OK` message indicates that all tests passed.

RUNNING A FAILING TEST

- What does a failing test look like?
- What can you learn from a failing test?

A test can fail if the code generates a Python error or if an assertion fails.

Here's a new test in *test_area_functions.py* that checks that `None` is returned for negative inputs:

```
    def test_negative_inputs(self):
        """Test negative inputs."""
        area = af.area_rect(-3, 4)
        self.assertEqual(area, None)
```

This test fails; here's the output:

```
$ python test_area_functions.py
F.
============================
FAIL: test_negative_inputs
Test negative inputs for area_rect().
----------------------------
Traceback (most recent call last):
  File "test_area_functions.py", line 15:
    self.assertEqual(area, None)
AssertionError: -12 != None
----------------------------
Ran 2 tests in 0.000s
FAILED (failures=1)
```

A failing test shows an `F`: the output shows the failed test and prints its docstring, the failed `assert` statement, and the `assert` statement's values. A final summary shows the number of tests, their runtime, and the failures.

FIXING A FAILING TEST

- How do you fix a failing test?

A failing test gives you the chance to fix a bug before it causes problems.

The function `area_rect()` doesn't pass the `test_negative_inputs()` test. To fix it, the function must do what the test output describes: if the area is negative, the function should return `None`. Here's a modified function:

```
def area_rect(w, h):
    """Calculate the area of a rectangle."""
    area = w * h
    if w >= 0:
        return area
    else:
        return None
```

Now the function returns `None` if the area is negative. Run the test case again:

```
$ python test_area_functions.py
..
----------------------------
Ran 2 tests in 0.000s
OK
```

Both tests pass.

Run tests every time you modify code and before releasing a new version to users. Also, add tests regularly to match your code updates.

THE SETUP() METHOD

- What does `setUp()` do in a test case?
- How can you use `setUp()` to make your testing code more efficient?

Place initial test requirements in the `setUp()` method to run automatically before each test to make code more efficient and consistent.

In the sample test case (Cards 8.2 and 8.4), we define widths and lengths in each test function. Instead, we can define width and length in `setUp()` and use the values in each test function:

```
class AFTestCase(unittest.TestCase):
    """Tests for area_functions.py."""

    def setUp(self):
        """Create widths and lengths for all tests."""
        self.width = 3
        self.length = 4
```

Then the test method would look like this:

```
    def test_rect_area(self):
        """Test a simple rectangle."""
        area = af.area_rect(
                self.width, self.length)
        self.assertEqual(area, 12)
```

This `setUp()` function is short. In a real-world test with complicated input values, the `setUp()` function makes test code more efficient and ensures consistency across many tests.

PACKAGES

One of the greatest benefits of programming in Python is the wide variety of software that's already been written, which you can use and build upon in your own work. You can install (and uninstall) most packages with just one or two lines of code. These cards show you how.

9.1 Installing Packages

9.2 Uninstalling Python Packages

INSTALLING PACKAGES

- What is a package?
- What is a library?
- What is `pip`?
- How do you install Python packages?

A *package* is a directory that contains modules. A package organized to be shared is often called a *library*. Installing a package gives you access to all its functions and code.

The `pip` tool installs Python packages and is included in all current Python versions.

To install a package with `pip`, run `pip` as a module with the package name:

```
$ python -m pip install --user requests
Collecting requests
--snip--
Installing collected packages: requests
Successfully installed requests-2.18.4
```

The `-m` flag runs `pip` as a module. The `--user` flag installs the package for the current user only. The `requests` argument is the package name to install.

You'll see the package download, output of any dependencies needed, confirmation of the installation, and the version.

UNINSTALLING PYTHON PACKAGES

- How do you uninstall a Python package?

You can use `pip` to uninstall any package:

```
$ python -m pip uninstall requests
Uninstalling requests-2.18.4:
--snip--
Proceed (y/n)? y
  Successfully uninstalled requests-2.18.4
```

You'll see a summary of related packages that will be uninstalled; enter `y` to proceed. A message confirms the uninstall. There's no need to use the `--user` flag.

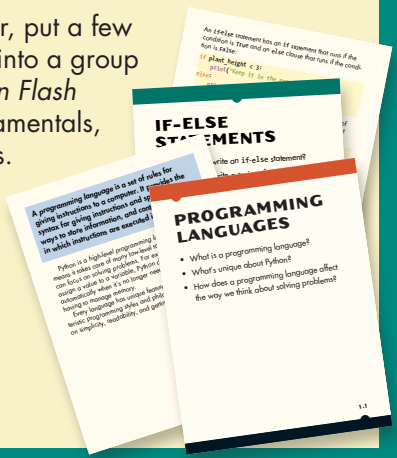
From the best-selling author of *Python Crash Course*

PYTHON FLASH CARDS

LEARN. REVIEW. TEACH.

The 101 cards in this box distill the basics of Python programming into bite-size pieces, together with plenty of code examples. Use the 34 concept cards to strengthen your understanding of core programming concepts like data types, code refactoring, and use of the command line. Refresh your Python knowledge with the 67 syntax cards, which cover topics like lists and dictionaries, control flow options, and functions and modules, as well as problem-solving techniques and best practices.

Whether you work through the cards in order, put a few in your pocket for on-the-go practice, or dig into a group of cards to master a specific topic, the *Python Flash Cards* are perfect for reviewing Python fundamentals, learning new skills, and even teaching others.



COLOR-CODED FOR EASY REFERENCE

no starch press

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

COVERS PYTHON 3.X • 101 CARDS