Python Bootcamp

Table of Contents

Module Introduction: Introduction to the Python Programming Language and Development Environments

- Watch: Features of the Python Programming Language
- Read: Summary of Some Key Python Resources
- Watch: The iPython Interpreter and Jupyter Notebooks as an Interactive Execution Environment and Program for Running Code
- Watch: Integrated Development Environments
- Watch: Introduction to Python as a Calculator
- Code: Manipulating Numerical Data Using Python
- Module Wrap-up: Introduction to the Python Programming Language and Development Environments

Module Introduction: Using Simple Python Data Types and Assigning Variables

- Watch: Introduction to Built-in Numeric Types
- Code: Investigating Different Types of Numbers
- Watch: Assigning Names to Data
- Code: Doing Mathematical Operations in Python Using Variables
- Quiz: Identifying Different Types of Variables
- Watch: Choosing Variable Names
- Module Wrap-up: Using Simple Python Data Types and Assigning Variables

Module Introduction: Using Python Container Data Types

- Code: Editor vs. Interpreter
- Read: Containers and Python's Built-in Types
- Watch: Introduction to Python Lists
- Code: Creating Lists and Assigning Them to Variables
- Quiz: List Operations: Valid Operations, Indexing into Lists

- Watch: Introduction to Python Dictionaries
- Code: Creating Dictionaries and Assigning Them to Variables
- Quiz: Dictionary Operations
- Watch: Introduction to Python Strings
- Module Wrap-up: Using Python Container Data Types

Module Introduction: Calling Built-in Functions and Methods to Manipulate Data

- Watch: Functions, Methods, and Namespaces
- Quiz: Relationship Between Functions and Methods
- Watch: Built-in Functions, Operators, and Keywords for Numeric Data Types
- Code: Investigating Built-in Functions and Operators for Numeric Data Types
- Watch: Built-in Functions for Container Data Types
- Read: Built-in Methods for Lists, Tuples, and Strings
- Read: Built-in Methods for Dictionaries and Sets
- Code: Investigating Built-in Functions and Methods for Container Data Types
- Code: Computing the Average (Mean) of a List of Numbers
- Tool: Built-in Python Container Data Types
- Read: Useful Features of the IPython Interpreter
- Tool: IPython Tip Sheet
- Module Wrap-up: Calling Built-in Functions and Methods to Manipulate Data

Module Introduction: Writing Custom Python Functions

- Read: Mechanics of Writing Functions in Python
- Watch: Defining Functions in Python
- Code: Write and Call Some Simple Functions
- Quiz: Return Values and Types From Functions
- Watch: Default and Keyword Arguments
- Module Wrap-up: Writing Custom Python Functions

Module Introduction: Using Basic Elements of Control Flow and Iteration in Python

- Read: Creating a Data Processing Pipeline With Control Flow
- Watch: Sequential Execution

- Code: Calling Functions Sequentially to Process Data
- Watch: Looping and Iteration
- Code: Iterating Over Lists and Dictionaries
- Quiz: Iteration Basics
- Watch: Code Blocks: The Role of Indentation
- Watch: Branching
- Code: Writing If Statements
- Watch: Exception Handling
- Watch: Abstracting the Process of Iteration
- Code: Investigating Iterables
- Read: A Review of Indexing and Comprehensions
- Read: Putting it All Together: Building Data Processing Pipelines
- Module Wrap-up: Using Basic Elements of Control Flow and Iteration in Python

Module Introduction: Creating New Custom Python Data Types With Classes

- Read: Object-Oriented Programming and Custom Data Types
- Watch: Bundling Data and Methods Using Classes to Create New Data Types
- Code: Building a Custom Class and Creating New Objects
- Quiz: Defining Classes, Including the Role of the 'self' Argument
- Read: Custom Data Types Provided by Other Python Libraries
- Module Wrap-up: Creating New Custom Python Data Types With Classes

Module Introduction: Crafting Numerical Calculations in Python

- Watch: Combining Python’s Functions and Data Types for Custom Calculations
- Read: Putting it All Together: Constructing Data Processing Pipelines
- Assignment: Build Custom Data Processing Pipelines
- Module Wrap-up: Crafting Numerical Calculations in Python

Welcome to Python Bootcamp

Video Transcript

Data science is a very exciting field, and one of the things that's exciting about it is that it encourages your creativity, to try to figure out how to gain insight and meaning from large datasets. But transforming your creativity into number-crunching routines requires expressive tools, and this is where Python comes in. Python is both an elegant programming language and a rich ecosystem of packages developed by a number of experts to support various kinds of data analyses and data visualizations. It's lightweight, yet it's powerful. So it lets you answer what-if questions posed by your colleagues almost as soon as they're asked. So, in this course, you'll get an understanding of some of the key features of Python that will let you begin to build data analysis pipelines.

Course Description

Data science is one of today's most in-demand functions — and it involves human construction of models to make sense of data. Mastering the ability to analyze and visualize data in meaningful ways using Python is a critical step in your eventual study of machine learning and artificial intelligence, which automate the process of pattern discovery and statistical modeling of data.

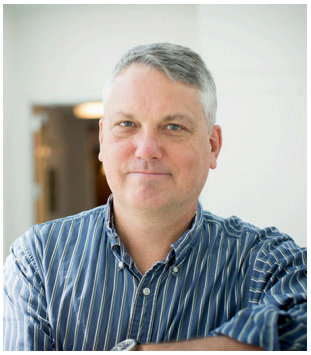
In this course, you'll start by exploring Python as a programming language and as an ecosystem of packages that support various kinds of data analyses and data visualizations. You will review and practice key features of Python that will allow you to build data analysis pipelines, and you will learn how to combine the built-in functionality of Python along with custom code to unleash the power of your data analysis pipeline.

What you'll do

- Articulate general design and features of Python and its relationship to other programming languages
- Use built-in Python data types, functions, and methods
- Assign and access variables
- Call built-in functions and methods to manipulate data
- Combine variables and data in expressions
- Leverage Python as a powerful calculator

- Write custom Python functions
- Use basic elements of control flow and iteration in Python
- Dive into Python container data types
- Create new custom Python data types with classes
- Crafting a Data Processing Pipeline in Python

Faculty Author



Chris Myers

Senior Research Associate and Adjunct Professor

Center for Advanced Computing

Cornell University

Chris Myers has been with the Center for Advanced Computing (CAC) since 2017, having previously been a member of the research staff of the Bioinformatics Facility of the Institute of Biotechnology (2007-2017) and the Cornell Theory Center (1993-1997, 1998-2007). In addition, Myers is an Adjunct Professor in the Department of Physics at Cornell and a member of the graduate faculty in the fields of Physics, Computational Biology, Applied Mathematics, and Computational Science and Engineering. Myers carries out research in the fields of complex systems and computational biology, addressing problems in the systems biology of cellular regulation, signaling, metabolism, development, virulence, and immunity; and in host-pathogen interactions and the spread of infectious diseases on populations, networks, and landscapes.

Course Resources

The contributing faculty and experts who designed this course have identified a number of additional resources that may be useful if you wish to delve further into the topics covered throughout the summer. You may choose to listen/read/watch any of these alongside your course work now or in the future.

AI Resources

- **A.I. Nation**: An NPR podcast that reveals how AI is affecting every major aspect of our modern lives.
- **Coded Bias Trailer**: A trailer for a documentary exploring the idea of technologies like AI being built on systemic racial and gender-based prejudices. The woman in the trailer is Joy Buolamwini, a computer scientist and digital activist. Buolamwini founded an organization, Algorithmic Justice League, to challenge bias in decision making software.
- **Kate Crawford's book**: Kate Crawford is a researcher at Microsoft research who recently wrote a book about bias AI. This video is a segment from one of her many talks where she promotes the importance of this topic.
- **Machine Learning and AI Roles at Google**: You can find information on various positions at Google that are in the ML and AI field.
- **Machine Learning and AI at Google**: Information on Google's AI research, responsibilities, and tools.
- **More AI at Google**: Google's official blog about AI.

Python Resources

The following are links to a variety of downloadable Python resources.

- **Core Python Data Science Ecosystem Components**: A tool listing core Python ecosystem components relevant to data science such as the Python Standard Library, IPython, Pandas, Numpy, and more.
- **Built-in Python Container Data Types**: A list of Python's built-in container data types such as lists and dictionaries.
- **IPython Tip Sheet**: Helpful tips for using IPython.

eCornell Keynotes on the Future of Tech

- **The Coming AI Revolution**: Keynote session hosted by Ray Jayawardhana, the Harold Tanner Dean of Arts & Sciences, as part of the Arts Unplugged series. An interactive discussion with leading experts on changes and considerations in how we can enact policy that supports democracy and an ethical society.

- **The New Jim Code**: Keynote session hosted by Ruha Benjamin, Professor of African American Studies from Princeton University, which discusses how technology, while appearing to be a neutral arbiter, deepens racial discrimination and ways in which technology encodes inequality and amplifies racial hierarchies.

Back to Table of Contents

Module Introduction: Introduction to the Python Programming Language and Development Environments

Python is a computer programming language that is widely used across a number of different application areas and has become a key tool in the field of AI. At its core, Python is a programming language that defines specific rules for how to instruct computers to carry out particular computations. But Python is also an environment or "ecosystem" that extends the capabilities of the core language into new problem domains such as data science.

Python is an interpreted programming language, meaning that statements are processed one at a time by a program known as an interpreter. This provides a high level of interactivity, where users can explore and interrogate data or prototype different sorts of analyses or data visualizations. There are multiple interpreters available for use with Python programs, such as the default python interpreter; the enhanced IPython, which is well-suited for interactive work; web-based Jupyter notebooks, which tie together code, analyses, documentation, and graphics; and integrated development environments (IDEs), which unite code editors, interpreter sessions, and data explorer tools.

Python's built-in functionality makes it an excellent tool to use immediately for carrying out numerical calculations. In this module, we will introduce you to both the Python programming language, and the IPython interpreter, which is well suited for scientific computations and is a good tool to use to practice Python and data science fundamentals.

Back to Table of Contents

Watch: Features of the Python Programming Language

Python is a powerful programming language that can serve a wide variety of needs. In this video, Professor Myers provides a high-level overview of Python and some of its key features.

Video Transcript

Python is a great programming language for doing data science, but of course there are lots of different programming languages and it's worth trying to understand some of the basic features of the Python language; perhaps allow you to compare your experience with other kinds of programming languages.

So Python is a general-purpose programming language, meaning that it wasn't developed to support some particular area. And as such, it's really very broadly applicable across a number of different areas, including data science. So this is in contrast, perhaps, to some other languages you might have some experience with, such as R, which was introduced as a statistical computing environment; or MATLAB, which has lots of support for linear algebra and matrix operations that arise in engineering contexts.

So because it's not targeted to a particular technical area, like data science, a lot of the useful functionality that we have in Python lives in external libraries; these are things that are written — that are not part of the core language itself but are part of packages and tools that we can bring into Python to get particular kinds of functionality, and this is a very important part of the extensions to the core language itself.

As a language, Python is interpreted, as opposed to compiled; what this means is that it it runs within a program, called an interpreter, that processes Python expressions, and in fact it will process expressions one at a time. And so this allows for very useful, interactive access to programs and data and interactive exploration of data, as opposed to what would — say in a compiled program, where we would need to write an entire program, run it through a compiler to get a working program that we could then — we could use to explore data. So this interpreted nature of Python makes it very useful for rapid prototyping development as well as data exploration.

Another key feature of Python is that it is object-oriented, which means that it provides a lot of support for defining novel and custom data types; not just the ones that are kind of built into the language itself. And this is very important for allowing us to capture various kinds of application-specific concepts or logic, or to develop more rich numerical environments for exploring data.

Python is a high-level programming language, meaning that it's both got a readable and clean syntax, and in fact this was an important part of its design from the outset; to make the intent of programs very clear just by reading them without actually having to run them. It's also high level in the sense that it has a number of powerful built-in data types that allow for very expressive and succinct calculations in less code than might be required, say, in some other programming languages.

In terms of its type system, Python is dynamically typed, meaning that we don't need to define the types of variables ahead of time. We don't need to know type declarations; essentially variables just acquire whatever type of object they're assigned to. And again, this is very well-suited for rapid prototyping and stands in contrast to a statically typed language where we need to define ahead of time type declarations and make sure that those are all consistent across our program.

And finally, Python is extensible, as are many languages, but extensible in the sense that we can integrate this high-level interpreted code with low-level compiled code written perhaps even in other languages, which is important for getting numerical efficiency and high performance in our data applications, our data analysis. So there's a lot of useful tools for extending the core functionality of the language with algorithms that are developed in other languages, and this extensibility is an important part of the language and we'll see all of these features in play.

Back to Table of Contents

Read: Summary of Some Key Python Resources

There is a wealth of information online to learn more about Python. The best place to start is the Python language website at **www.python.org.**

In this course, you will be working with Python version 3.

> ☆ **Key Points**
>
> You will be working with Python version 3.
>
> You can learn more about Python at **www.python.org**.

Accessed through the tabs at the top of the Python.org site, there is also:

- Documentation describing the language and a tutorial to help get started (**https://docs.python.org**)
- Information about the Python user and development community (**https://www.python.org/community**), including pointers to Python conferences and special interest groups you might like to attend
- A link to the Python Package Index (**https://pypi.org**), a centralized repository of available Python software supporting a bewildering variety of tasks
- Information about open job postings for people with skills in Python (**https://www.python.org/jobs**)
- Information about the Python Software Foundation (**https://www.python.org/psf-landing/**)

Although the Python.org site also contains links to Python distributions that you can download and install on your own machine, it is generally easier to install Python instead from some other source. This is because there are several "batteries included" bundles available for download that include not just the core Python distribution but also many third-party libraries useful for data science. If you are interested in installing Python and associated libraries locally, see our separate page on this topic.

Back to Table of Contents

Watch: The iPython Interpreter and Jupyter Notebooks as an Interactive Execution Environment and Program for Running Code

There are a variety of different programs that can interpret Python code, and each is best suited for different use cases. The default Python interpreter is great for running full programs that will process data and produce suitable outputs. The enhanced IPython interpreter provides additional functionality for interactive access and data exploration. Jupyter notebooks represent an even richer environment that merges code, results, graphics, and documentation into live documents.

In this video, Professor Myers illustrates how Python interpreters provide interactive execution environments that enable you to run and test your code in real time.

Video Transcript

All right; well, Python is a programming language, but it's also a program that interprets that programming language. It's a little confusing; sometimes the language is written with a capital P and the program that interprets it is written with a lowercase p. So this interpreter, the Python program, is good for short calculations and command-line batch jobs that are going to run in background, whereas the enhanced IPython interpreter provides a lot of useful additional features to support interactive work.

So what do we mean by an interpreter? An interpreter is a program that runs on your computer. It's a program that your operating system knows about. But what it does is sit there and wait to be fed Python source code. So we can either enter that in, say, one line at a time or we can write a program and feed it in an interpreter. The interpreter makes sense of those instructions that you've provided and then it produces output; this could be text or graphics or other kinds of data. And again, this is to be contrasted with a compiled program that we have to compile all at once before we can run.

So as was mentioned, the IPython interpreter is a useful interactive tool, and I'm showing you some of what that looks like here. I've entered a couple of expressions where I can, say, add some numbers or print "Hello" or multiply some other numbers. But I can also, for example, ask about information about different types of data that I might be interested in creating or printing other kinds of expressions in this interpreter. So, by having an interactive session where I can enter Python code and execute it and look at results, this is very useful for certain kinds of data science applications.

There's actually an even more powerful extension of this core IPython interpreter known as Jupyter Notebooks. And this then goes beyond just having this code interpreter, but it's a more

rich environment than integrates code and results as well as graphics and other kinds of visualizations, and documentation and text that we use to describe our analysis. So instead of being run in a terminal window like the IPython interpreter, Jupyter Notebooks run within a web browser. And in fact Jupyter is a very general framework that supports interactive computing for a number of different programming languages; not just for Python.

So this is kind of what a Jupyter Notebook looks like here. We've got this web browser that's got documentation. We've got some headers that we are writing. We can write this documentation in a markdown language. So we can have text describing what we're calculating. We can have code where we're importing and executing various kinds of Python functions. And then we can, for example, plot data; we can generate data and plot it and integrate the graphics into this live document that is pulling together all of our code, all of our analyses, all of our documentation, into one live computing document.

So code can be run in a number of different ways. We can either type it directly into the interpreter as we did before, or into a Jupyter Notebook, and this is good for interaction, interactivity, and exploring data. But sometimes we want to be able to rerun analyses over and over again, and so there it's very useful to write our code separately in a text file using an editor and then running it through the interpreter where we can then maybe change parameters or if we want to repeat these analyses over and over again. And so these different execution frameworks, modalities that are provided with Python make it very useful for running anything from very little scripts that compute something very simple to long-running programs doing complicated data analyses that might be running for days on multiple processors on a high-performance cluster.

Back to Table of Contents

Watch: Integrated Development Environments

Writing programs and carrying out data analyses in Python requires the integration of various tools, such as editors to write source code, interpreters to run that code, and perhaps additional programs to convey results. Depending on your work preferences and the tasks at hand, you might prefer to keep those tasks separated or to have them bundled together in an integrated development environment (IDE) or a Jupyter notebook. In some cases, there are efficiencies that derive from having all the necessary tools at hand, much like the way a chef chooses to organize all the pots, pans, and utensils in a kitchen to ensure easy access to the right tool for the task at hand.

Video Transcript

So when we're developing code, we're really engaging a set of collaborating tasks. We're using editors, for example, to write Python source code — and it's useful to have an editor that understands Python syntax to assist you with that — and then we're using interpreters to interpret that source code. Now, in principle, those processes can exist independently of each other and just be brought together in your workspace, but often it's useful to work in what's called an integrated development environment, or an IDE, which combine, in a single program, both an editor and an interpreter, as well as adding perhaps other sorts of functionality for browsing files or for inspecting the values of different variables.

So this is an example here that I'm showing you of an IDE called Spyder, and on the left-hand side there's an editor window in which I've defined a simple function, f(x,y), that returns the product of x and y. And then on the upper right, I have an IPython interpreter window, which is waiting to — it's sitting there waiting to get commands. And so, for example, I can run this function — this code on the left that defines us this function f, and now within my interpreter I can define, say, a few variables, x and y, and I can call the function f(x,y) and get the result of the product; 12 and 3 — it's 36. So that's running in the interpreter.

Down below in the lower right, we actually see in this variable explorer window that the variables x and y now have been added along with their types and their values, and it's keeping a record of what we've been doing within the session. If we, for example, wanted to assign the value of this expression, this product f(x,y), to some other variable z, we could do so, and now we can see down below in the variable explorer that z has been added as well. So this is a useful environment for allowing us to write code, to run and interrogate the results of that code, and then also to have access to the variable — the various pieces of data that we've been creating and analyzing.

So, the IDE is one type of environment that allows you to integrate these different tasks. Jupyter Notebooks are another example that allow you to integrate code and analysis and documentation. But those Jupyter Notebooks are sometimes not so good for development; they're good for running and demonstrating — documenting analyses at the end, but for the process of developing new code, new algorithms, new analyses, it's often more useful to work in an environment like an IDE where you can have this integration of an editor, an interpreter, and other sorts of tools.

In some sense, it's personal preference; there's different ways to solve these kinds of problems. But it's also a question of suitability for tasks. So, you might — for certain kinds of tasks, you'd be using an IDE to develop code, and then maybe later in a workflow using a Jupyter Notebook to show those results to collaborators or co-workers, of course.

In this and subsequent videos, Professor Myers will be using the Spyder IDE. For more information, visit the Spyder website: **https://www.spyder-ide.org**. In this program, you will be working in a similar IDE to practice Python and data science fundamentals. You will then work in the Jupyter Notebooks environment for the remainder of your work in data science and machine learning.

Back to Table of Contents

Watch: Introduction to Python as a Calculator

You've probably used some type of calculator before — such as a physical calculator, one on your phone, or in a spreadsheet, or the software calculator that comes with computers.

As you'll discover in this video, you can also use Python as a calculator. In fact, as Professor Myers explains, using Python as your daily calculator is one of the best ways to quickly get up to speed in understanding Python.

Video Transcript

So Python is a very powerful programming language but it's also incredibly useful just in a much simpler use case, which is as a calculator. I use Python as a calculator many times on a daily basis rather than going to some other app or machine to do such a thing. And that's because there's basic support for arithmetic operations and expressions which, of course, arise in all sorts of different contexts, both in terms of business and, say, personal finance if we're trying to calculate monthly expenses or taxes on purchases or compound interest; what have you.

So, within the IPython interpreter, we have access to all of this mathematics. Just again, all the basic operations are available, so if we want to add a couple of numbers, we can do that. And we're inputting 2 + 3, and then the output associated with that is, of course, the number 5. We can subtract two numbers, we can multiply two numbers. Or that wasn't a multiply. So we can just go back; we can actually re-edit the previous line in this interpreter and go back up and change the plus to a star and get multiplication. We can divide two numbers; 2 divided by 3. We can exponentiate, so 2 to the third power. We have access to a double-star operator, so that's, of course, 8.

So, when I did 2 divided by 3, I got the fraction 0.6666 repeating. If I say 23 divided by 3, I get the fraction 7.6666. But sometimes we want to do not what is called floating point division like this, but integer division. So if you think back to when you were in grade school, and if you were given a division problem 23 divided by 3, the answer you would come up with is 7 remainder 2, and that's an example of integer division and we can get that with not just a single slash but double slashes. So 23//3 gives us the integer part of that division, which is 7, and then 23 mod 3 gives us the remainder 2. So, 23 divided by 3 is about 7.6666 as well as 7 remainder 2.

We have access to other sorts of mathematical operations. We can, for example, round numbers to the nearest integer, whether it's the nearest integer up or the nearest integer down. We can, of course, string operations together, so — and this is where it's important to know about operator precedence. So, for example, if we say "2 + 3 * 4 + 1", you might be wondering, well, what order are things executed in? And in fact the multiplication operation takes precedence there. So

that would be equivalent to, for example, if we had put parentheses around the 3 and the 4 and executed that first; that gets done before the other two additions. We can, of course, explicitly use parentheses to group various kinds of operations. So, "(2 + 3) * (4 + 1)" gives us 25.

The exponentiation operator, like the multiplication operator, also takes precedence over addition and subtraction. And if we wanted to, for example, make sure that we were grouping things correctly, then we can again use parentheses to force that kind of operator precedence. So again, we can group various calculations together. We can combine them to do kind of nontrivial things.

So, for example, let's look at this piece of code here where I've got some items that I'm purchasing, some of which are taxable — they're going to be some tax charged on those — and some of which are nontaxable. And if I want to, for example, load this into my environment here, I can load in this source file I've defined, and what we see is that I can combine the set of taxable items, which are defined above, and tax them at this tax rate of 8%, and then add them to this set of nontaxable items to find my total cost of this purchase, which is $50.85.

So, Python is a calculator; it's an example that you often see in documentation as a simple way to learn Python, but it's also incredibly useful for manipulating data, and the sooner you get going using Python and IPython as your daily calculator, the faster you will be in getting up to speed in using Python for data science.

Back to Table of Contents

Code: Manipulating Numerical Data Using Python

Manipulating numerical data is central not just to data science but likely many of your daily activities, whether that involves shopping, banking, or helping someone with their homework. A calculator is a familiar tool, but carrying out involved calculations using one can be slow and cumbersome. Python provides powerful support for carrying out mathematical operations on numbers and may become the mathematical tool that you turn to once you get the hang of using it as a calculator.

This exercise is not graded. It is designed to familiarize you with various arithmetic operations in Python, such as addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**), as well as grouping of terms and subexpressions in order to build up more complicated expressions. You don't need a code editor for this because you will simply execute a sequence of expressions directly within the interpreter, examining the output of each. It's good to be able both to write programs in a code editor and use the interpreter interactively for short calculations.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Introduction to the Python Programming Language and Development Environments

As you have seen, Python is a powerful programming language that can be used for many purposes. In this module, you've examined how Python interpreters provide interactive execution environments that enable you to run and test your code in real time. You have also explored integrated development environments and how to use Python as a calculator. Next, you'll look at data types and variables.

Back to Table of Contents

Module Introduction: Using Simple Python Data Types and Assigning Variables

Like most programming languages, Python provides several built-in numerical data types, such as integers, floating point numbers, and Boolean values (`True` and `False`). Python also supports the ability to assign data to variables, that is, names that are associated with data objects. There are rules that constrain how variables can be named; understanding these rules will allow you to both write easy-to-read code and to interpret possible errors that might arise if those rules are not followed.

Back to Table of Contents

Watch: Introduction to Built-in Numeric Types

As Professor Myers explains in this video, Python has a wide variety of numeric types you can use in your programs. Knowing what these types are will help you understand how Python interprets them.

Video Transcript

So as with most programming languages, Python has a set of built-in numeric types. These are different types of numbers that have different kinds of properties. And of course we know this from mathematics; we have things like the number line, the real numbers which can involve rational and irrational numbers and so on; and then we have the integers, for example, which are at discrete locations along that real number line. And so Python provides support for different types of numbers as well as a built-in function called type() that we can use to query the types of different objects.

So we have the integers, which are called ints; floating point numbers are real numbers which are called floats; complex numbers, which you may have encountered in some of your mathematics courses which involve both the real and imaginary part of the complex plane. There are Boolean values, true and false, which are often important in use in deciding logical statements. True and false, or these bools, are, in fact, special cases of ints where true is identified with being as equal to 1 and false is identified to being 0. But it is a special separate case of its own.

And all of these things, these numerical data types, if I speak of the integer +3, or the integer -2, or the floating point number 1.5, these are all literals. These are all called literals in a programming language because they are just literally what they are. It's not like they're somehow masking for something different than what they are. So literals are an important concept in programming languages, and all of the numerical data types in Python are literals.

So we have lots of arithmetic operations to manipulate these different types of data, and we can interrogate the types of information or types of variables within the interpreter. So here we are back in the IPython interpreter and I can do my favorite operation, 2 + 2, since I always remember what the answer is, so that gives me 4. But of course I can also ask, "What is the type of 2 + 2?" and that tells me that it's an int. So type() is a built-in Python function that takes an object — in this case, the thing that is returned by the expression 2 + 2 — and it tells me what type it is.

If I, for example, now add an integer and a float — so 2 + 2.0 — this gives me the floating point number 4.0. So there is an integer of 4, of course, but now there's a type conversion that when I add an integer to a float, Python interprets the result as a float which is a more general construct

than the integer. So I can ask, for example, what is the type of this thing that I just created, 2 + 2.0, and I see that it's a float.

I can, of course, do my other sorts of numerical operations like exponentiation, like 2 to the third power. I can ask about the type of that; "What is the type of 2 to the third power?" It's an int, but of course now if I take 2 and raise it to the minus third power, that's equal to 1 over 2 to the third, and so that's a floating point number and that is, of course, confirmed by the result when I ask the type of object.

So there's these different data types and they can get converted among each other as a result of the kind of mathematical operations that are being carried out, and type() is a useful function for figuring out what any particular object — what its type is.

Back to Table of Contents

Code: Investigating Different Types of Numbers

Like most programming languages, Python provides support for different types of numbers, such as integers, floating point (real) numbers, complex numbers, and Boolean variables. Manipulating numbers is at the heart of data science, and understanding the different number types in Python is central to that effort. Operations between numbers of one type sometimes produce a result of a different type. For example, if you add, subtract, or multiply two integers, you always get an integer as a result. But if you divide two integers, you usually get a floating point number (float) as a result. The type of any number can be investigated using the built-in `type` function. For example, `type(12)` will tell you that 12 is a number of type `int` (integer). It's very useful to know how to identify the types of numbers and possibly convert their type based on how they are entered or read into a program.

In this exercise, you won't write code in a code editor, but you will get some practice within the interpreter examining the types of different numbers.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Assigning Names to Data

While Python enables you to do mathematical operations directly with numbers, you can also assign those values to variables, which are names attached to objects. You can then execute the same mathematical operations with variables that you would do with the numbers themselves.

In this video, Professor Myers explains how assigning names to data helps makes code more readable and easier to manipulate.

Video Transcript

So we have all this great access to arithmetic operations to carry out calculations, but with all these numbers kind of floating around on the page, it gets a little hard to keep track of everything that's going on. And so an important part of really any programming language, and certainly Python, is the ability to assign names to data. And those names are important for several reasons, one of which is that by giving names to values, we facilitate being able to read the code but also being able to change the values of those variables if circumstances change.

And so we can see this, for example, in the interpreter that we have a set of purchases, some of which are taxable and some of which are not, and we've entered in both the set of purchases that are taxable and not, as well as the tax rate. And now if we want to compute the total, we can form an expression; rather than just having a big, long arithmetic set of numbers and operators, we can write an equation. We can say the total is equal to the taxable amount times 1 plus the tax rate — this tax rate is 8% — plus the nontaxable amount. Of course, the nontaxable amount doesn't have any additional tax associated with that. And so now when we execute that, we can, for example, interrogate the total and we can see that it is $50.85.

And of course all of these variables that we've defined in the interpreter window up above show up down below in the variable explorer window, and so we have access to those. But as I said, one of the nice things about having things written such is that if we decide that we want to — say circumstances change; that our local government decides that we're not taxed enough and so they want to increase the tax rates. So the tax rate now goes up to 9% and we now need to recompute how much those purchases are, so we can just re-execute that same expression we had before and we can see, for example, that now the total has gone up to $51.05.

So, by giving names to data, we both create more readable code, but we also allow ourselves the ability to be able to make changes to the values of things and then rerun analyses again with that updated information.

Code: Doing Mathematical Operations in Python Using Variables

While Python is very useful as a calculator just to work with numbers, as a programming language, it is much more powerful than that. Assigning data to variables is central to programming, allowing for calculations that are not hard-coded to specific numerical values, and that are more easily readable because variables can be given meaningful names. In addition, by assigning data to variables, you can reuse those variables in different parts of your calculations.

As with the last few exercises, you will work directly in the interpreter to carry out various tasks involving variables, their types, and the results of operations on variables. You will assign data to variables, query the types of those variables, and carry out arithmetic operations on variables.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Identifying Different Types of Variables

Let's test your knowledge of different types of variables.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Choosing Variable Names

Assigning data to variables can make code more readable and reusable, but it is useful to choose variable names judiciously to assist in that process. Constraining the choice of variable names are a set of rules that Python imposes about what sorts of variable names are valid. As Professor Myers explains, this is just one example of why choosing variable names is an important skill for any Python programmer.

Video Transcript

So we can assign names to variables in Python and that's very useful, but an important matter for consideration is how to choose variable names. There are both rules about what sorts of names we can construct as well as good practices in terms of how to name things. So let's talk about the rule. So Python has syntactic rules about what constitutes a valid variable name. A name must start with a letter or an underscore, so it can't start with a number. It can have numbers within the name; the remainder of the name can consist of letters or numbers and underscores. But it can't have spaces or other kinds of special characters in them. Variable names are case sensitive, so if in one case the name is capitalized, now they're not, those are going to be totally distinct variables sharing no information.

And as a matter of good practice — I mean, variable names are not like passwords; when you want to make a password, you want it to be unbreakable and unguessable, but the whole point of assigning names to your data are that you're helping to make the code readable both for you and potentially for others. So, again, there are rules about valid names. So if we look in the interpreter, maybe I'm describing a circle and I want to say that the radius of the circle is, say, 12 centimeters. So I can type that and that's fine, and now "radius" has appeared in my variable explorer as a valid variable. But, for example, let's say I forgot about the rules and I wanted to call it "the radius"; I said "the radius" equals 12 centimeters. I get an error here, and in fact we see this SyntaxError that I'm using invalid syntax, and the invalid syntax here is that I put a space between "the" and "radius" and that's not allowed. And the reason it's not allowed is that spaces are used to kind of separate information. I mean, we've seen this with arithmetic expressions, that spaces are used to separate variables and data and expressions, and so to have a space in the middle of a name makes it very hard for the interpreter to know, is this one thing, is this two things; what have you.

There are other rules. If we decide we really want to call it "the radius", we can, of course, put an underscore in the middle of the name. So "the_radius = 12.0". And now we see in the variable explorer below that that has been added as a new variable. But let's say we don't like the look of underscores and we want to maybe stick a hyphen in there instead. And if we try that, we get

another SyntaxError, although now the error isn't that this is invalid syntax; it says "can't assign to an operator". So the operator that we're trying to assign to is the subtraction operator, the hyphen; this is saying we want to subtract "radius" from "the" and assign its value to 12, and Python says you cannot assign a value to an operator. So it's often important to not just get frustrated when you see an error message like this but actually read the error message, see what it's telling you, and use that to understand, for example, how you might have constructed a bad name.

So I mentioned that there was case sensitivity, so if I say, "Radius = 14" now with a capital R, we see in the variable explorer that we now have not just "Radius" with a capital R whose value is 14, but we also have "radius" with a lowercase r whose value is 12. And again, it's sometimes useful to use capitals versus lowercase names in defining different variables, but you want to make sure that you're not confusing one for the other in your code.

There are other syntactic rules. There are a set of reserved keywords that Python defines; these are words that are central to the construction of the Python language, and you're not allowed to assign to any of those. So one of them, for example, is the keyword "def" that we use to define functions. And so if I try to define — say set "def = 3", it'll tell me, again, this is invalid syntax; the reason that this is invalid is because this is a reserved keyword that I'm not allowed to overwrite. We don't want to be trashing parts of the Python language just so we can assign some value to it. "lambda", for example, is another keyword, and we get the same kind of error when we try to assign to that. So these reserved keywords are kind of off-limits; even though the word "lambda," which is a Greek letter, is sometimes useful to include in mathematical expressions, it has an important role in the Python language and so we have to find some other name instead.

So the reserved keywords are things that we absolutely cannot assign to, but there are built-in functions that we really should not assign to because unlike the keywords, which Python just says, "No, you can't overwrite that," with these built-in function names we actually can override them, and that would be bad. So, for example, float() is a function that if we give it an integer, for example, it converts that into a floating point number, a different type of number. But if I decided I wanted to assign to the name "float" the value 12, if I now try again to convert the integer 3 to the floating point number 3, I get a different type of error. Now I get this error that says an int is not callable, because basically I've taken this very useful function float() and I've overwritten it by the number 12. And so you really need to be careful about how you define things; you need to be aware of the rules for what's allowed, what's good practice, and how not to step on yourself when you're writing code.

Python rules for valid variable names:

- A name must start with a letter or an underscore.
- Remainder may consist of letters, numbers and underscores, but not spaces or special characters.
- Names are case sensitive. Variable name `hello` and `HELLO` are different variables.
- Do not use Python's reserved keywords or built-in types and function names as variable names. For more information on Python's reserved keywords **visit Section 2.3.1 of the Python Reference Manual**.

Back to Table of Contents

Module Wrap-up: Using Simple Python Data Types and Assigning Variables

In this module, you saw how Python has a wide variety of numeric types that you can use in your programs. Knowing what these types are helps you understand how Python interprets them. In addition, you explored assigning names to data and how this makes code more readable and easier to manipulate. You also practiced mathematical operations in Python using variables and examined how to choose variable names.

Back to Table of Contents

Module Introduction: Using Python Container Data Types

In addition to built-in numerical data types, Python also provides several built-in container data types. As the name suggests, containers are data objects that hold other data objects. A list that holds a sequence of numbers is one common example of a container found in data science, as is a string that holds a sequence of textual characters. Containers comprise an expressive set of tools for manipulating collections of data and are an important element in Python's ease of use. Understanding what different container types are used for, as well as the commands by which their data are accessed and changed, is time well spent.

Back to Table of Contents

Code: Editor vs. Interpreter

Up until now, you have been entering and executing Python statements one at a time in the interpreter and examining the results. This is very useful for a variety of tasks, such as doing small calculations, querying data that you have loaded, or testing out a small bit of code to do an analysis or make a plot. For larger calculations, such as those involving many steps or those that you will want to run repeatedly, it is better to write Python code in an editor and then run the code within the interpreter once you're done writing it. When you're running a program file in the interpreter, each statement still gets executed one after the other, but the entire set of statements will get executed as part of that process.

The way you interact with your data is a little bit different in these two different scenarios. When you run a program, you usually want to produce some output. What would be the point of running a long program that calculated a bunch of things but then exited at the end without saving the results of any of those calculations? One method for producing output that you have already used is the built-in `print` function, which prints a textual description of an object or variable to the screen; there are similar sorts of functions for writing that information to a file that we will investigate later.

Throughout all these exercises, you will sometimes be instructed to do something directly within the interpreter, and sometimes instructed to write code in the editor so that it can later be run (either by you or by the system, in order to check your output). Even if you are writing code in the editor, you should always feel free to try things out in the interpreter to test some operation or verify that you understand what a particular function or expression is doing. Learning Python by testing code out interactively is very useful, even if your ultimate goal is to write more complex code in an editor to carry out longer calculations.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Read: Containers and Python's Built-in Types

> ☆ **Key Points**
>
> Containers are used to bundle sets of related data together.
>
> Different container types bundle data elements in different ways and are intended for different purposes.

Containers, also known as collections, are composite data types that — as the name suggests — contain other data types. They are useful for bundling together sets of related data rather than having separate variables for each individual data element. Think of a shopping list in which you write down everything that you want to purchase at a grocery store. You don't carry around a separate piece of paper for each item but instead put them all into one container. You then iterate through the items in the list so that you can get everything you need. Containers simplify both the process of storing sets of related data items and the process of operating on them.

Different container types bundle data elements in different ways and are intended for different purposes. Containers are common to almost all programming languages. In Python, there are several containers that are built in as part of the language and others that are accessible in additional libraries that you can access from within Python. Some of the key built-in Python containers are:

- **lists:** for storing a sequence of items in a specific order
- **dictionaries:** for associating unique members of one set of items (keys) with members of another set (values)
- **sets:** for storing a group of unique items in no specific order
- **strings:** for storing a sequence of textual information (e.g., to produce words, sentences, paragraphs)
- **tuples:** like lists, for storing a sequence of items in a specific order, but cannot be modified once created

Operating on data containers lies at the core of data science. A data scientist using Python needs to first understand how these core built-in container types operate since they are both useful in their own right and form the basis of other, more complex containers provided by external libraries.

Back to Table of Contents

Watch: Introduction to Python Lists

Sometimes you might want to group specific items in a program.

For example, perhaps you are writing a program to manage inventory for a grocery store. You might want to create groups such as:

```
fruits = ['apples', 'bananas', 'oranges']
```

```
vegetables = ('carrots', 'broccoli', 'bok choy')
```

Did you notice how the fruits are grouped with brackets and the vegetables are grouped with parentheses? These are examples of lists and tuples, both of which Professor Myers explores in more detail in this video.

☆ **Key Points**

Python lists:

Group objects in specific order

Useful when ordering of objects is important

Can contain different types of Python objects (including other lists)

List items:

Are ordered sequentially and indexed by position in list (starting at 0)

Can be accessed by their position

Can be reassigned by their position (lists are mutable)

Lists can be sliced and concatenated

Video Transcript

So Python defines a number of useful container classes that are used to hold or contain other types of data in useful groupings, and a very important container in Python is the list object. So a list groups a bunch of Python objects in a specific order, and so it's useful when the ordering of those objects is important. And lists are very flexible, they can contain all sorts of different types of different objects — not just necessarily one type of object — so they can hold mixtures of ints and floats and character strings and even other lists.

So the lists provide a number of useful operations; they can be indexed, essentially queried as to what exists at different positions in the list. Those items can be assigned to, and there are a number of other useful operations that are defined. So because the order of items in a list is important — they are ordered sequentially and they're indexed by their position in the list. And in Python, as in many other programming languages, that counting starts at 0. So the 0th element of an array in the list is the first item that you encounter in the list, and the thing at Index 1 would be the second position, and so on. So items can be accessed by their position and they can be assigned by their position. So lists are mutable; they can be changed. This is an important feature

of lists. So even if we have an existing list, we can reassign some element by accessing it at its position.

We can get access to contiguous groups of items — not just individual items, but groups of them in a chunk — through an operation called slicing, and then we can also concatenate the lists with the plus operator, and we'll see all of these within the interpreter. So here we are in the interpreter, and we can define a couple of lists, say, named list1 and list2. These are just objects that we can — lists we can define by embracing — including a set of objects in square brackets. And once I define these, I can see down below in the variable explorer that these objects, list1 and list2, have been created. As you can see, list1 just has a set of integers in it; whereas list2 has a heterogeneous mixture of different types of information, including containing a list itself.

So we can access various elements in the list. We can ask for the 0th element of list1 and find the integer 1. We can ask for in list2 the item at Position 3 and we get this list "1, 2, 3, 4," which by virtue of the counting starting at 0 is in the last element of the list named list2. We can do slicing — well, we can do assignments. So, for example, if we wanted to change the value of this item at Position 3 in list2, we can reassign that just by assigning to its position. And now if we look at what list2 consists of, we see that it no longer has this list in its last element but in fact has the integer 100 that we assigned to.

Slicing gives us the ability to access larger chunks of data within a list, and so let's create a longer list now called list3. And what slicing does is allow us to grab sublists within that list. So, for example, if I want to slice out from Positions 1 through 5, what this does is it will pick out starting at Position 1 and continuing one short of Position 5. So it'll include Positions 1, 2, 3, and 4, and that, in this case, happens to equal the values 1, 2, 3, and 4, but in this list of ordered integers. So by slicing different elements of a list, we can get access to pieces of it, say. We can't do that because we have to actually access it on the list; not on the list() function but on the list3 object.

And we can, in fact, concatenate different lists with the plus operator. So we used the plus operator previously to add different numbers, for example, but we can also use the same plus operator to add two lists; in this case, taking all of the elements for list1 and concatenating at the end of them all of the elements from list2 and returning a new, bigger list which is this concatenation.

So that was lists, which are a mutable sequence of objects in Python. There are also tuples, which are very much like lists but which are immutable; that once we create them, they cannot be changed. And so tuples, instead of using the square bracket notation, are constructed by enclosing a set of items in parentheses such as I've done here. So I've created two new tuples, tuple1 and tuple2, that contain the same data that we found in the previous lists we defined. We can slice these things the same way; we can say, "What is the 0th element of tuple1?" and get the

integer 1. But now if we want to try to reassign that value, we want to change its value to something else, we see that we cannot do that; that a tuple object does not support item assignment; that once we've created the tuple, it can no longer be modified. I can create a new tuple, for example, by concatenating two tuples, but I can't modify one of the elements of those things themselves.

So Python provides us lists and tuples for managing ordered sequences of objects, and they both play important roles in building up more complex programs.

To learn more about Python lists and how to access data in them, visit **section 3.1.3 of the Python Tutorial**.

Back to Table of Contents

Code: Creating Lists and Assigning Them to Variables

Containers are objects that hold other objects. They are extremely useful for bundling together groups of related information and for applying calculations to all the elements that are held in the container. Python has several built-in containers that serve different purposes. The first one we will look at is the `list`. You probably use lists frequently in your daily life to help keep things organized: you make a shopping list, a to-do list, a list of people you need to contact, etc. Maybe you put the items in those lists in a particular order, or maybe you just add things in any order, such that the order doesn't matter as long as you visit each of the elements in the list. In Python lists, the ordering of items does in principle matter, in that they can be accessed by their position in the list.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: List Operations: Valid Operations, Indexing into Lists

Let's test your knowledge of list operations.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Introduction to Python Dictionaries

Think about a traditional dictionary. Traditional dictionaries match terms with their definitions. In Python, you might call the term a “key” and the definition a “value.”

Now think about how useful this type of relationship (keys and values) can be in a Python program. In this video, Professor Myers explains what Python dictionaries are and how to use them.

Video Transcript

All right; well, another extremely important container data type within Python is the dictionary, and this is used to group Python objects together that have a relationship where we want to associate some value with some key, some sort of identifier. So it's a way of associating one set of data with another, and we call these things key-value pairs; they're pairs of keys and values together that — one of which the key implies in association with the value.

So we can create dictionaries in Python using curly brackets; not using the other kinds of brackets we've seen for other objects, but using curly brackets. And they're a bit like lists in that we can index into them to get out particular items from a dictionary. But rather than indexing with an integer index like we do with lists and tuples, we index by the key. And we'll see this in the interpreter.

☆ **Key Points**

Python dictionaries:

Group Python objects by mapping keys (k) to values (v)

Associate one set of data with another (k-v pairs)

Are created with curly brackets

Are similar to lists but identified by user-specified keys rather than an integer index indicating position

Key-value pairs appear in no guaranteed order

Within dictionaries:

Entries are accessed via square brackets

An item can be accessed by its key

An item can be reassigned by its key (dictionaries are mutable)

Additionally:

Many Python objects are valid as keys, but not mutable containers like lists and dictionaries

Dictionaries cannot be sliced like lists (no order)

So one of the important things about dictionaries is that the key-value pairs are in no defined or guaranteed order. So although one can visit every element in a dictionary, they're not necessarily going to arrive in any particular order, and so you should never write code that depends upon them being in a particular order.

So within the interpreter, we can see some of this — we can see some of this happening. We can create, for example, a dictionary, dict1, and what I've created here is that I've initially entered three key-value pairs. So I'm associating the value 1 with the key a, I'm associating the value 2 with the key b, and I'm associating the value 3 with the key c. I can add — dictionaries are mutable, so I can add new elements to this, for example. I can add the key d with the value 4, and I can add the key e with the value 5. And now if we look at what dict1 consists of, it's got all of these — now five key-value pairs that we've defined, and you can see this down in the variable explorer as well.

So we use the same square bracket notation that we saw for lists to index and reassign elements in dictionaries, and we can in fact reassign because dictionaries are mutable; we can overwrite values to entries. So, for example, we can reassign the value — the key C to be something different than what it was originally associated with. So now, for example, we associate with this long character string, and if we look at dict1 we see that now the key c is associated with the character string "abcdefg" rather than as was previously with the value 3.

So dictionaries provide a number of useful operations. Although we can't, for example, slice into a dictionary — like we can't slice items out of a dictionary like we have with lists. And in fact we want to remember that when we index these things, we don't index so we can — we index by the key, So dict1 of a returns the value 1. If we forget and think that this is a list and we try to index by an integer, what we'll see, for example, is a KeyError that we've asked for Entry 0 in dict1, but if we look back at our dictionary, we see that there are no keys named 0, and therefore we get an error from the interpreter saying there's no key in this dictionary whose identity is 0.

So, again, dictionaries are a very useful container data type; not where there's a sequence of elements but where there's a set of key-value pairs that are associated with each other.

Back to Table of Contents

Code: Creating Dictionaries and Assigning Them to Variables

Dictionaries are another important type of container in Python. Whereas Python lists contain a sequence of objects in a particular order, dictionaries (or dicts, for short) define a relationship between a set of keys and a set of values. Think of something like the set of contacts on your phone or computer. This is a data structure that takes the name of a person and produces a set of associated information (phone number, email address, etc.). If you want to call your Aunt Mary, you don't need to remember that she is entry number 129 in your contact list and look her up that way; more conveniently, you can look her up by her name. In this example, "Aunt Mary" is the key in a dictionary, and her phone number is the value associated with her name. For example, if your contacts were not stored in a dictionary but instead in a pair of associated lists called contacts_names and contacts_numbers, you might do something like this to retrieve Aunt Mary's phone number:

```
AuntMary_position = contacts_names.index("Aunt Mary")
# AuntMary_position will be 129 since that is her position in the names list
phone_number = contacts_numbers[AuntMary_position]
```

Whereas, to access the phone number associated with the key 'Aunt Mary' in a contacts dictionary, you would simply write:

```
phone_number = contacts_dict["Aunt Mary"]
```

In both cases you are getting items out of containers using the square brackets operator. The mapping of keys (names) to values (phone numbers) is more straightforward using a dictionary. And as it turns out, it is also quicker to use a dictionary since looking up a key in a dictionary is faster than looking up an item in a list. This is a widely used data structure that goes by a variety of names in different programming languages: dictionary (in Python and .NET), map (in C++ and Java), associative array (in awk and PHP), and hash (in Perl).

With a list, you access an element by its position number. With a dictionary, you access an element by its key. For a list, the position numbers must be integers starting at 0 and ending at N-1 (for N elements in the list). For a dictionary, keys can be pretty much anything, such as integers, floats, strings, dates, etc. (There are some types of objects that cannot be used as keys, but we won't worry about that now.) But otherwise, accessing and setting elements is similar, using the square bracket operators in both cases.

In this exercise, you will practice some of the basics of creating dictionaries, accessing their elements, adding new elements, and resetting their values.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Dictionary Operations

Let's test your knowledge of dictionary operations.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Introduction to Python Strings

The last Python container type we'll explore in this module is strings.

Strings have their own unique characteristics, but they also share some characteristics with lists and tuples. Strings are a key type of container in Python used to represent textual data. They are a bit like lists in that they represent a sequence of objects in a particular order and are manipulated in some similar ways. For example, characters in a string can be accessed based on their position in the sequence using square brackets, and two strings can be concatenated using the `+` operator. Strings are unlike lists in an important way, however: while lists are mutable (i.e., their elements can be modified), strings are immutable. Once a string is created, it cannot be changed, although the information contained within a string can be accessed and combined with other information to create new strings. For example, the expression `MyName = "My" + "Name"` concatenates the strings "My" and "Name" to produce a new string assigned to the variable `MyName`, but it does not alter the two strings that are joined.

☆ **Key Points**

Python strings:

Are ordered groups of textual "characters"

Can contain letters, numbers, whitespace, unicode characters, etc.

Can create with matched pairs of single, double, or triple quotes (allows for embedded strings)

Can be indexed and sliced like lists

Are immutable like tuples

Can be concatenated with the + operator

Professor Myers introduces Strings in this video.

Video Transcript

So Python provides support for character strings, namely an ordered group of textual characters that allow us to build up pieces of text, as do many other programming languages. These strings are containers in their own right and they support many of the same sorts of operations that we see in other Python containers. So we can build up strings out of — that contain numbers and letters and whitespace and Unicode characters, and we basically can create them using either matched pairs of single, double, or triple quotes. And this is useful because then it allows us to create embedded strings; we can have one string — say, a quotation — within a larger text string.

We can see this within the interpreter. We can create, for example, these two strings, string1 and string2. In string1, I'm embedding the phrase 'Hello, world!' within a matched pair of single quotes; whereas in string2, I'm using the outer pair of double quotes to define the overall string. And within that I have a sub-string, which is matched with single quotes. So when I create these strings they show up in the variable explorer below. And as was mentioned, there's also the opportunity for creating using triple quotes, and this is useful for creating multiline strings. So I've entered in this new string, string3, which says, "This is a multiline string." And if I look at the value of this thing, we see that it's the string "This is a multiline string", but it has embedded in it this newline character, this /n which is used to represent that line break there. So that's sort of technically what the string consists of, but if I ask the string3 to print itself, we actually get back the character return as part of the multiline string. And these triple-quoted multiline strings are often useful in defining documentation strings within Python code.

So as a container that's got an ordered collection of objects, we can index into a string, just like we do with lists and tuples. We can ask, "What is the 0th element of string1?", and that is the capital letter H. We can slice elements of strings. We can say, "Give me Elements 0 through 5." Slicing stops one short of the upper bound of string1 and that gives me this sub-string "Hello,". I can slice bigger chunks if that's something that I'm interested in. I can slice out the sub-string from string2, "'Where are you going?'" And as with lists and tuples, I can concatenate strings. So for example, I can slice out part of string1 and part of string2 and put them together to create a new string, "Hello, 'Where are you going?'"

So the same kinds of operations that we see for lists and tuples as word collections are reproduced with strings, but it should be noted that strings, like tuples, are immutable. So once I create a string, I can't reassign some part of it. So, for example, if I want to redefine what's at Element 3 in string2 with, say, the letter C, I get this error, and we're told that a string object does not support item assignment. So, like a tuple, once I create the string, I can manipulate it to create new strings, but I can't go in and change an element of that string itself.

Back to Table of Contents

Module Wrap-up: Using Python Container Data Types

In this module, you examined containers and Python's built-in data types. You took a look at some of the most commonly used container types, lists, dictionaries, and strings. In addition, you had the chance to practice creating and assigning variables to these containers.

Back to Table of Contents

Module Introduction: Calling Built-in Functions and Methods to Manipulate Data

Programming is essentially the act of constructing a data processing pipeline: the state of a program (that is, the values of the data it contains) is transformed through a series of functions. Together the program state and its transformation form the basis of a grammar, much the way that nouns and verbs do in human language. Python contains many built-in functions for transforming different data objects. Understanding the logic of function calls and method calls is a key to understanding Python programs. Central to this logic is the notion of namespaces, which organize what functions can act on what types of data objects. Everything in Python is an object, including both the simple numbers and the functions that transform them, and every object has an associated namespace that dictates what it can and cannot do.

Back to Table of Contents

Watch: Functions, Methods, and Namespaces

In this video, Professor Myers lays the groundwork for three of the most critical aspects of Python: functions, methods, and namespaces. Together, these three aspects form the foundation of reusable code.

☆ **Key Points**

Python functions:

Take inputs and produce outputs

Can also change inputs

Python methods:

Are functions that are attached to objects

Some methods alter the object they are attached to

Some methods return a different object

Indexing on lists and dictionaries as methods

Video Transcript

So a central part of any programming language is the ability to define functions that transform data, and in an object-oriented language like Python, we're often also defining methods, which are essentially functions that are attached to objects rather than existing as kind of stand-alone operations. So functions are generically things that take inputs and produce outputs, and they can also, in some cases, change what those inputs are, although that can sometimes lead to some confusion. Methods, as I said, are functions that are attached to objects, and they often are accessing internal data within those objects in order to transform that somehow. Some methods might return a different object. Some methods might just alter the object that they're attached to internally. And when we looked, for example, previously at indexing into lists and dictionaries, those accessing and slicing operators are themselves methods that are defined on those types of objects.

So let's see a little bit of what this looks like within the interpreter. So I've defined a simple list here, list1, whose values are 6, 5, 4, 3, 2, 1. And one of the things that Python provides is a built-in function len(); I can ask, for example, "What is the length of list1?" by calling the function len(), and it tells me 6; this is telling me there are six elements in list1. I can also attach to the object list1; there is an append method. And so, for example, I can append a new element to that list; append the element 0. And now if I look at list1, I see that I now have my original list plus the 0 that's been appended at the end. So that append is a method that's defined on the object of type list, whereas this len() is a function that exists as a built-in function within the interpreter.

So, another built-in function is sorted(); I can say I want to sort the elements of list1. And what sorted() does is it takes the list that I pass it and it returns a new list which is a sorted version of

that list, but it does not, for example, alter the original list, which we can see here; that list1 still has this original order, but sorted(list1) returned a new list with a different order. If, instead, I call the method sort on the object list1, this does not return anything; as is seen within the interpreter, there is no output associated with that call there. But now if I query list1, I see that now it has, in fact, been reordered by this sort operation. So there are two complementary functionalities, one of which is sorting a list to produce a new list, and another of which is calling a sort method on an existing list to reorder it internally.

So, these functions and methods, depending on what's being called, are controlled by what are called namespaces. And a namespace is essentially — you can think of it as a folder for where things reside. So, for example, len() and sorted() are functions that live within a built-in namespace that the default Python interpreter defines, whereas sort and append are methods that live in the namespace attached to the list object. And the reason for having these namespaces is it allows us to control the complexity of code; that we don't just have one big bundle of names that all are sort of piled on top of each other. But by separating things into different namespaces, we can identify sort of subsets of code that are important for particular operations.

So, these built-in functions, like len() and sorted(), are part of the built-in namespace, and then the various methods that we call are part of the namespaces with their associated objects. So not only do lists have a set of methods attached in their namespace, but dictionaries do as well, and those execute operations on the data within the dictionary. Other functions might reside in other namespaces in optional modules that we can import. And in all of these cases, we access items in a namespace with the dot operator. So when I typed list1.sort(), what I was basically saying is, "Go access the sort method that's attached to the list1 object," and this is a way that we can essentially open up that folder and get at the operation that lives within it.

So, this gets to sort of a much bigger idea, is that essentially everything in Python is an object. I mean, all the data types we've looked at — containers, functions, methods, modules — everything is an object and everything is in a namespace and every object has an associated namespace, and we can often query these to find out what kinds of operations different functions, methods, and objects are capable of. So many operators that we encounter, such as the arithmetic operations — plus and minus and so forth — the square bracket indexing operator, the len() built-in function, these are all associated with methods defined on different objects. And if we, for example, asked what's in — the dir() function is a built-in function that tells us all the methods associated with some object. If I say dir(list), I get a list of all of the different functions that are attached to list; some of which are these things that we've already encountered, such as append and sort, as well as kind of hidden and built-in things such as len() and add() and getitem(), getitem() being the function that gets a particular item out of a list. So

it's important to understand that everything is an object, everything has a namespace, and that you can learn about what the capabilities of different objects are by querying their namespaces.

Back to Table of Contents

Quiz: Relationship Between Functions and Methods

It's time to test your knowledge of functions and methods.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Built-in Functions, Operators, and Keywords for Numeric Data Types

As Professor Myers explains in this video, sometimes you might be working with numeric data types and want to manipulate them in certain ways to achieve your programming goals. For example, you might have an integer, such as 4, that you want to convert to a floating point, such as 4.0. Or you might want to convert a numerical value that's stored as a string into a numerical data type. Some common type conversion functions that you will use are `int`, `float` and `str`. The function `str` is especially important if you would like to print a numerical value. For example: `print(str(4))`.

Video Transcript

So Python defines numeric data types as part of its built-in construction, but there's also a number of built-in functions and operators that allow us to manipulate those data types. In addition to the arithmetic operators, there are a number of mathematical functions that we can use to examine numeric data types. So within the interpreter, for example, we can see that we can calculate the minimum — the min of 2.1 or 2.5, which will return the smaller of those two items; in this case, 2.1. We can ask, for example, "What is the absolute value of -3?" and get the number +3. We can round floating-point numbers to integers. And these are just a few of the built-in functions that are meant to act on numerical data types.

There are also important relational and logical operators that are defined that allow us to query the relationship between different data types. So, for example, within the interpreter, I might define these variables w, x, y, and z, and then I can ask about various relationships. So if I ask, "Is x equal to y?", well, x and y are both 3, so that is, in fact, true. If I say, "Is x greater than y?", this will tell me that this is false. So both this double-equals and this greater-than are operators that take, in this case, two operands — x and y in this case — and they return a Boolean true or false depending on the relationship of those variables. I can ask, for example, "Is x greater than or equal to y?" And that is also true, because they are equal. I can even construct more complicated sorts of operations, and that I can, for example, add x and y, and I can add w and z, and I can ask whether those two things are true. So, x plus y is 6, w plus z is 6, so the comparison of those two things is, in fact, true.

So these kinds of logical operators are often used to control the flow of data processing in a program, and we can use these to branch in our calculations based on whether, say, two values are equal or not. We can also manipulate data using built-in functions to either create or convert data between different data types. So, for example, the int operator is something that creates an integer, and if I pass it a floating point number, what it does is convert that floating point to an

int by essentially chopping off the decimal part of it. So, even if this were, say, int(3.9), this would still be converted to 3 because it's not a rounding operation; it's a chopping operation. Int is very smart; it can also take the string, quoted string 3, which is a string variable, and convert that to the integer 3. But it can't, for example, take the string 3.1 and convert that into an int, because 3.1 — it can't do that double conversion; it cannot convert that string, which essentially is — a string is a floating point number to an int.

So we can, of course, create floats in other directions; we can create a float from a string — say we can create a float 3.1 from the string. We can create a bool from the value 3.1. A bool is actually a true-or-false, and so anything that's non-zero evaluates to true when converted to a bool, but a bool of zero, say, is false. So with these kinds of built-in functions for converting between different data types, querying different numerical data types, and comparing them, we have a rich set of operations for building up more complicated calculations.

Back to Table of Contents

Code: Investigating Built-in Functions and Operators for Numeric Data Types

You have already used some of the basic arithmetic operators (+ - * / **) to carry out some mathematical operations on data. Python has additional built-in functions for manipulating and querying numerical data, beyond those basic operations. Some of these are functions that enable you to compute some properties of numbers or groups of numbers. Some of them are additional logical operators (== > <) that let you ask logical (True/False) questions about the relationship between two numbers, such as "Are two numbers equal?" or, "Is the first number greater than the second?" Other functions perform basic mathematical operations on numbers, such as computing the absolute value of a number or rounding a floating point number to the nearest integer. Having these in your toolbox will assist you in crafting data analysis pipelines.

You'll be working directly in the interpreter to investigate the results of different operations, and remember that the help function is also a useful tool when you want to understand what a particular operation does.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Built-in Functions for Container Data Types

Just as you can query and manipulate numerical data types, Python provides similar functionality for container data types. In this video, Professor Myers provides examples of functions you can use to work with Python container data types.

Video Transcript

So Python provides a number of built-in container data types such as lists and dictionaries and sets and tuples and strings, and it also provides a set of built-in functions for querying and manipulating those data types. And so these involve both built-in functions for data creation and conversion, say creating a list from data or creating a dictionary from data, as well as querying and manipulating those.

So let's take a look at a few examples in the interpreter. We can create, for example, a string from the float 3.1 using the function str(). This converts the floating point number to the string str, which is, as was noted, a type of container. We can create a dictionary, either directly by specifying all the set of key-value pairs in curly braces, but we can also, for example, define a set of key-value pairs separately; in this case, as a list of pairs, and the pairs are encoded as tuples. So the key-value pairs — ('a', 1), ('b', 2), ('c', 3) — can be used to create a dictionary; in this case, dict2, which, as we can see in the variable explorer, contains the same information that's in dict1. And so this is very useful; sometimes we want to be able to build up information to create a data type through one mechanism, say by building up this list of pairs, and then at a later stage convert that into a dictionary, rather than having to build it into the dictionary all by itself at once.

We can create, for example, a list, either directly using the square bracket notation, or we could build it through other means. But then we can, for example, convert that list to a set. So if we use the set() function acting on a list, we now have created a set that has the elements 1, 2, 3, 4; whereas the list that we created from had the elements 1, 2, 3, 1, 1, 2, 4. The reason that the set has fewer elements is because a set can only contain unique elements, and so all of the duplicated 1s and 2s that were in the list didn't get double-counted when we produced the set. But sometimes that's a useful operation, to be able to construct that sort of reduction.

We can create a tuple from the information that's in the list, and what that does is essentially convert our mutable list with these items into this immutable tuple. There are built-in functions that allow us, for example, to query the length of a list; we've seen this before. There are seven elements in list1. But the same operation knows how to figure out what the length of a set is. And so the length of this set1 is 4 because of this reduced number of elements that were of unique items that were included.

Some of the same operations that are defined on lists, for example, like list — like max, I can ask, "What is the maximum element" — "the largest element in list1?" That is the number 4. That same max operator works, for example, on the set object, set1, returning the largest element there. I can sort the elements of a list; we saw this previously. I can sort the elements of a list in numerical order. I can also sort the elements of the set in numerical order. But we see now that this, instead of producing a set, returns a list, as noted by the square brackets here. And the reason that this is, is because a set intrinsically — the items in a set have no intrinsic ordering. And so if we're asking explicitly to sort the elements of a set, we are essentially defining an ordering, and therefore we're converting that to a list. So there's sort of an implicit conversion from a set to the list when we expose it to a certain type of operations such as a sort.

There is membership testing; we can ask, "Is a given item in a list?" So, 3 is, in fact, in list1 because — let's remember that. 3 is, in fact, also in set1, and that set lookup is very fast. But, for example, if we were to look at our dictionary, dict1, and we were to ask, "Is 3 in dict1?", we'd see that that's false. And the reason for that is, is that membership testing for dictionaries is essentially done on their keys — In this case, a, b, and c — and not on their value. So even though the value 3 is in the dictionary dict1, it's not a member of the keys, and therefore it is not — that operation returns false.

So there are a number of built-in functions and operations that act rather generally across different container data types where it makes sense to implement the same kind of functionality across different data types, such as asking how many elements are in the array — or in the object, or what's the largest element in a list; those kinds of things are supported across a number of different kinds of container classes in Python.

Back to Table of Contents

Read: Built-in Methods for Lists, Tuples, and Strings

Lists, tuples, and strings in Python are all ordered collections or containers, and they each define a set of methods on each of these data types. In some cases, there's some shared functionality across these different container types because they are all ordered collections.

Methods for Lists

Methods for lists include:

☆ **Key Points**

Some methods for lists will throw an error depending on the contents of the list and what is being requested.

Methods that query an existing tuple are supported.

Like lists and tuples, strings support indexing and slicing via square brackets [ ] and concatenation via addition +.

Methods for Lists

METHOD	DESCRIPTION	EXAMPLE
append	Adds element to end of list	my_list.append(5)
index	Returns first index (position) of element in list	your_list.index(10)
count	Returns number of times an element occurs in a list	his_list.count('c')
sort	Sorts list in-place by altering the data in the list	her_list.sort()
reverse	Reverses order of list in-place	my_list.reverse()

In some cases, the methods above will throw an error depending on the contents of the list and what is being requested. The index method, for example, throws a ValueError if you ask it to return the position of an element that is not in the list. Similarly, the sort method throws a TypeError if the list contains different types of items that cannot be meaningful compared to one another (such as ints and strings).

In addition to methods that we can call on lists, there are a number of operations that are supported, such as indexing and slicing using the square bracket operator [ ] and concatenation using the addition operator +. These same operations are available for tuples.

Methods for Tuples

Because tuples are immutable container types, however, methods that alter a tuple are not supported. But methods that query an existing tuple are supported.

Methods for tuples include:

Methods for Tuples

METHOD	DESCRIPTION	EXAMPLE
index	Returns first index (position) of element in tuple	my_tuple.index(11)
count	Returns number of times an element occurs in a tuple	your_tuple.count('d')

However, we can't append, sort, or reverse a tuple.

Methods for Strings

Strings are sequential orderings of a particular data type. While some programming languages distinguish formally between single characters and strings that are collections of characters, Python does not do so. Even a single textual character, such as the letter "A" is considered a string in Python; it just happens to be a string of length 1. Therefore, a string is a container that holds elements of type string.

Because strings are sequentially ordered containers, they share some similarities with lists and tuples. But because strings are immutable (like tuples), there are some sorts of methods and operations that they do not support. And because strings contain elements of a particular data type, they support some specialized methods that do not make sense for more general containers.

Like lists and tuples, strings support indexing and slicing via square brackets [ ] and concatenation via addition +. Strings also have an index and count method to find the first position of a specified element or the total number of times an element occurs in a string, respectively.

Like tuples, strings do not allow appending, sorting, or reversing because they are immutable.

But strings define some additional methods that apply to the specific type of string data that they hold. There are many such methods; examine the output printed by help(str) to see the full list and their descriptions. A few such methods are described in the table below.

Methods for Strings

METHOD	DESCRIPTION	EXAMPLE
isdigit	Returns True or False depending on whether the string is made up entirely of digits	'1234'.isdigit() # returns True
islower	Returns True or False depending on whether the string is all lowercase	'ABCdef'.islower() # returns False
isupper	Returns True or False depending on whether the string is all uppercase	'ABCDEF'.isupper() # returns True
lower	Returns a new copy of the string converted to lowercase	'ABCdef'.lower() # returns 'abcdef'
upper	Returns a new copy of the string converted to uppercase	'ABCdef'.upper() # returns 'ABCDEF'
capitalize	Returns a new string in which the first character is uppercase and the rest lower	'hello'.capitalize() # returns 'Hello'
startswith	Returns True if a string starts with a specified prefix, False otherwise	'Python'.startswith('Py') # returns True
endswith	Returns True if a string ends with a specified suffix, False otherwise	'filename.csv'.endswith('.csv') # returns True

So to conclude, lists, tuples, and strings are all sequential orderings, but because they have slightly different uses and functionalities, they define both a subset of common operations as well as some distinct operations that are applicable to each of them separately.

Back to Table of Contents

Read: Built-in Methods for Dictionaries and Sets

Dictionaries and sets also come along with some built-in methods that can be operated on objects of those types, but because dictionaries and sets don't have any intrinsic ordering, certain kinds of operations don't make sense.

> ☆ **Key Points**
>
> Dictionaries and sets also come along with some built-in methods.
>
> The sets objects in Python define useful set operations.

Methods for Dictionaries

Important methods for dictionaries include:

Methods for Dictionaries

METHOD	DESCRIPTION	EXAMPLE
keys	Set of all keys used to map values in a dictionary but in no guaranteed order	my_dict.keys()
values	Set of dictionary values in the dictionary mapping	your_dict.values()
items	Set of all key-value pairs in the dictionary (each pair is a tuple)	my_dict.items()
get	Returns the value associated with a certain key, but can also return a default value if that key does not exist in the dictionary	your_dict.get(k, 0)
pop	Returns the value from the dictionary associated with a key and removes (or pops) the key-value pair from the dictionary	my_dict.pop(k)
update	Updates existing dictionary with the keys-values from another dictionary	this_dict.update(other_dict)

Methods for Sets

The sets objects in Python define useful set operations.

Methods for Sets

METHOD	DESCRIPTION	EXAMPLE
union	Returns new set as union of this set and another set	my_set.union(your_set)
intersection	Returns new set as intersection of this set and another set	my_set.intersection(your_set)
difference	Returns new set with everything in this set that is not in another set	my_set.difference(your_set) # same as my_set - your_set

Back to Table of Contents

Code: Investigating Built-in Functions and Methods for Container Data Types

We've already started exploring the different container types that Python provides, such as lists, dictionaries, and sets. We'll continue to investigate them further to understand how some of Python's built-in functions can be used with data stored in container types. Having these functions in your toolbox will assist you in crafting data analysis pipelines. In addition, we'll see that some of the same functions can be applied to different types of containers.

In this exercise, you'll write code in the editor window, but of course you can always test things out or look for documentation in the interpreter too.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Code: Computing the Average (Mean) of a List of Numbers

Data analysis often involves building up calculations by combining various operations on data. Calculating the mean value of a group of numbers is a good example of such functionality. Because Python contains many built-in functions, we can use those in our calculations rather that building up all our code from scratch, resulting in more compact and readable code, written more quickly and probably more correctly. Let's investigate this by calculating the average value of a group of numbers: we can compute this by adding up the values of all the numbers and then dividing that total by the number of elements in the group.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Tool: Built-in Python Container Data Types

Container data types are essential elements in the Python programming toolbox. Understanding what different containers are used for, as well as how to call functions and methods to access and manipulate data in containers, is key to leveraging the power of Python for a variety of data science applications.

Back to Table of Contents

Download the Tool

This helpful **Built-in Python Container Data Types** tip sheet can aid you in using the appropriate container data types in Python.

Read: Useful Features of the IPython Interpreter

Python is a programming language with a defined set of rules, keywords, and built-in objects and functions from which you can build up programs and data processing pipelines. But Python is an interpreted language, which means that a separate program (called an interpreter) is required when you actually want to run Python code that you have written. While the programming language adheres to a specified set of rules, there exist different interpreters that can augment support for the Python language with additional features to support your work.

☆ **Key Points**

IPython is an alternative interpreter for Python that's useful when you want to explore your data and try out new analyses.

IPython provides magic functions that are useful in an interactive context but aren't recognized by the default Python interpreter.

IPython forms the core of Jupyter notebooks.

The default interpreter for Python is a program called "Python" (or possibly "python.exe" on a Windows system). That program understands only the Python programming language and is especially useful for running whole programs that you have written in a code editor and want to run from the command line. But if you want to do interactive work with Python, where you might load some data, inspect it, do some preliminary analyses, or make some plots, it is useful to have an interpreter with some additional features to support that kind of interactivity.

This is where IPython comes in.

IPython is an alternative interpreter for Python, intended to support interactive work. IPython is the interpreter that we are using for exercises in these courses. With enhanced support for accessing help and documentation, command-line editing, and special "magic" functions, IPython is especially useful when you want to explore your data and try out some new analyses.

It is important to keep in mind, however, that some of the features that IPython provides — especially the magic functions — are not part of the Python language itself. Magic functions in IPython begin with a percent sign, `%`, such as in the magic functions `%ls` (to list files in the current directory), `%run` (to run a Python code file in the interpreter), or `%timeit` (to time how long it takes to execute particular operations). And while these functions are useful in an interactive context, they will not be recognized by the default "Python" interpreter that you might use for command-line, Python-only programs.

Here are some other useful built-in IPython functions and magic functions you can use in the IPython interpreter:

- IPython provides several ways of getting documentation about datatypes and functions.
  - The built-in `help()` function: prints out help about an object; run `help(object_name)`. You can also use the `help()` function to get information on general types, e.g. `help(list)`, `help(dict)`, `help(int)`. To exit the interactive help system, type `q` and press enter or return.
  - You can also get information about an object by typing `?` after the name of an object or function, e.g. `list?` will provide short documentation about that object.
  - The built-in function `dir()` prints out just the group of names of methods (and potentially other attributes) associated with an object, without all the documentation that `help()` provides; run `dir(object_name)`. You can also use the `dir()` function to get information on general types, e.g. `dir(list)`.
- IPython magic functions are very useful for querying and manipulating data in the interpreter.
  - `%lsmagic` provides a list of available magic functions.
  - `%run` allows you to run your code in a source file within the interpreter.
  - `%history` allows you to see your input command history, and `%pdoc` prints the documentation text (docstrings) associated with an object, e.g. `%pdoc list`.
  - If you are familiar with a Linux/Unix environment, you will notice that many of the magics correspond to familiar shell commands (with % prepended), such as cat, cd, cp, less, ls, man, mv, and pwd. These commands enable you to manipulate and interrogate files on your local machine. Type `%ls` to list files in your current directory. This can be very useful if you are trying to read in a file or run some code, and want to check that the file is where you think it is. If you're not sure what directory you are in, type `%pwd` to print the working directory.
  - Another useful pair of magics are `%who` and `%whos`. `%who` describes what variables have been created in your main namespace and %whos provides summary information about those variables.
  - If you want more information about a particular magic, you can just type the name of the magic and append ? to it. For example, `%cd?`.

IPython forms the core of another interpreter that you will learn more about later, namely Jupyter notebooks. Jupyter notebooks extend the interactive access provided by IPython with additional ways to integrate graphics and documentation alongside your code. You can use the built-in IPython functions and magic functions in Jupyter Notebooks.

Back to Table of Contents

Tool: IPython Tip Sheet

Use this tool as your cheat sheet to IPython and magic functions.

For full online documentation about IPython, visit this link: **https://ipython.readthedocs.io/en/stable/index.html**.

For full documentation on magic functions, check out this page:

https://ipython.readthedocs.io/en/stable/interactive/magics.html.

Back to Table of Contents

Download the Tool

Use the **IPython Tip Sheet** as your guide to IPython.

Module Wrap-up: Calling Built-in Functions and Methods to Manipulate Data

You've explored the foundation of reusable code (functions, methods, and namespaces) as well as the relationship between functions and methods. In this module, you saw how built-in functions and operators allow you to manipulate data types and examined some built-in functions and methods for different container types.

Back to Table of Contents

Module Introduction: Writing Custom Python Functions

In this module, you'll examine the mechanics of writing functions in Python. You'll explore how to write and call functions as well as how to use default and keyword arguments in functions. Furthermore, Professor Myers will explain how to use docstrings in your code.

Back to Table of Contents

Read: Mechanics of Writing Functions in Python

This process of creating functions is often similar across many different programming languages, although there can be significant differences between languages based on what sorts of information a particular language might require.

> ☆ **Key Points**
>
> In general, a function takes one or more inputs and produces an output.
>
> In Python, a function definition consists of three parts:
>
> - prototype or header
> - function body
> - return statement

Broadly speaking, a function takes one or more inputs and produces an output. (The output can consist of multiple pieces of data, so it could also be said that a function takes one or more inputs and produces one or more outputs.) In that sense, a function can be thought of as a "black box" whose internals might not be entirely known to you, but you can still write a program that uses the function by understanding what kinds of data need to be input and what kinds of data will be output.

In Python, a function definition — which is identified through the use of the `def` keyword — consists of three parts:

- a prototype or header that specifies the name of the function and the names of the function's inputs
- a function body that processes those inputs
- a return statement that indicates what is output by the function

The return statement is part of the function body, in that it is indented in the code block under the line with the `def` keyword.

When the Python interpreter processes this function definition (which is typically a set of statements spanning multiple lines of code), a new function object is created and can then be called.

The function definition specifies the names of a function's inputs. These are just internal names used in the function definition and do not need to correspond to the names of variables in your program that are being passed to a function. The whole point of a function definition is to encapsulate a set of operations so that they can be reused with many different inputs.

Back to Table of Contents

Watch: Defining Functions in Python

Defining new functions in Python is accomplished using the `def` keyword. This enables us to specify the inputs of a function, the outputs of the function, and any data processing that takes place within the body of the function.

Video Transcript

So essentially, all programming languages provide some sort of mechanism for defining new functions and Python is no different, and the central keyword that arises in the Python language to allow you to define new functions is the def keyword. Short for define, def. So, when you use the def keyword, you're essentially signaling to your program that you're defining a function, you're defining a new function, and as with most functions that are developed in programming languages, there are going to be set of inputs that are provided as arguments to that function, and a set of outputs that are returned.

So, in Python, the way we structure this is as I said with the def keyword, there's a function name, then maybe say, "func" if we're not being very imaginative, if we may define a new function called func, and this function may take in this case, three arguments. Three different variables that are going to get to come into the function, we're going to process somehow, and then return some sort of output. So, we have the function header, which includes to this def keyword and the function name and set of arguments, and then a function body, which is indented with inside the function definition, which carries out whatever kind of calculation we are trying to carry out. Then returns some value whatever it is that to be carried out by this function.

Now, this is all rather abstract, so we could consider a couple of simple examples, we might define a function called add, that takes two arguments that we're going to call x and y. These names that we're giving these things are arguments are just how we're going to refer to these variables internally within the function. They may be called something else outside of the program went and this is called. But in this add function, we're going to take two variables x and y, and we're going to compute their sum by adding x plus y, and then we're going to return this up. That's all this function does is it takes two things that can be added together and returns their sum.

Or you might define something like an absolute value function "abs" which mathematically takes a number, and that number is greater than zero, then it returns greater than or equal to zero returns the number itself, and if it's less than zero, then it returns the negative of that number. So, that the absolute value always returns a positive number. This logic is captured in

this function definition we had here, if x is greater than or equal to zero, then we return x. Otherwise, return minus x.

So, by writing simple functions that do one thing and do one thing well, we can start to both manage the complexity of larger programs and also reuse code where necessary. So, it's important to remember that you're typically returning a value from a function, and sometimes when you're getting started with programming in Python, you might forget a return statement. So, a Python function without a return statement is valid, it's just that it returns what's known as the object none which is sort of on a null value. So, sometimes you may write a function and you think that it is supposed to work correctly and then instead you find that none as somehow being returned, and this is probably because you've forgotten a return statement.

It's also important member that functions can modify the arguments that come in. That in its cleanest form, they take inputs and produce outputs but they don't actually modify those inputs in place, but they can in fact that sometimes happen. So, it's important to document that clearly, or to provide an option to encourage users to remember that there is going to be some in-place modification taking place. Because that is sort of not to default type of behavior that functions often have.

Back to Table of Contents

Code: Write and Call Some Simple Functions

Writing and calling functions is a central process in programming, both for breaking complex calculations into more manageable units and for enabling efficient reuse of code. Identifying repeated operations in code and writing functions to capture those operations is a key element in programming.

In this exercise, you will work both on identifying opportunities for code reuse and on writing Python functions to implement those repeated operations. You will also encounter conditional statements (if/else), which specify which block of code gets executed depending on whether a condition is true or false. This concept will be revisited later in this course, but suffice to say that they are control flow statements that determine which path of execution a program will follow.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Return Values and Types From Functions

Now it's time to test your knowledge of the values and types returned from functions.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Default and Keyword Arguments

The simplest function definitions in Python involve positional arguments, where each input is mapped to an argument based on its position in the group of arguments and all inputs must be provided. Python also allows us to define default and keyword arguments. Default arguments assume a specified default value if an input is not provided to override that default. Keyword arguments are specified by both their name and their value. In this video, we consider some of the uses of these alternate forms of function definitions and function calls.

Video Transcript

So, when we define functions in Python, we provide possibly a set of arguments as inputs and in their simplest construction we have to provide all of those arguments, the variables to fill all those arguments slots when we call a function but in some cases it's useful to be able to have default values that we don't always need to specify explicitly and Python provides support for this default specification of a variable.

We'll see this in this live session here, we can say, for example, define a function here called exceeds_threshold. So, we're using the def keyword to define a function called exceeds_threshold and it takes two arguments, one of which is a value, the other which is a threshold and what this function does is it returns true if the value exceeds a threshold, otherwise it returns false. That seems rather straightforward. As we would normally have to define this, we would have to provide or to call this, we'd have to provide a variable both for value and for threshold. But as you see in the function specification here, I've given an extra specification I said threshold equals zero. So, what this does is it says the default value for this threshold and this function is zero, which means that I actually don't need to specify it and if I don't specify it, then the value of the threshold will be zero.

So, I can, for example, say exceeds_threshold(2) and this returns true because the input value of two exceeds the threshold zero. But if I explicitly provide a threshold, say 10, then this returns false because two is not greater than 10. So, this is useful when there's a useful default value that makes sense to set, it's not always the case that it makes sense to define some default value or it's not clear what the best default value should be and it's worth pointing out that many of the built-in functions in Python also often use default value. So, for example, there is a sum function in Python, it's a built-in function that as we see from the help, returns the sum of a start value plus an iterable of numbers. So I might, for example, provide a list of this 1, 2, 3 and the sum of 1, 2, 3, 6 and that's because it added each of those elements 1, 2, 3 to the start value, the default of which was zero. But, for example, I can call this with a different start value, say 100, and now the sum is 106.

So, there's functionality that's there that's almost hidden and sometimes you need to, for example, read the documentation to know that's in fact it and that you can access that so you don't always have to provide, you don't have to say type in all of these arguments if there's a useful default argument. So, the inputs to functions can be fried either positionally as I've done here in these examples or by what are called keyword arguments. So, with keyword arguments, the inputs don't need to be passed in this specific order and this is often very useful when you're trying to override a default value in a set of longer input arguments.

So let's go back to my exceeds_threshold, for example, and I can in fact call these things in a different order as long as I give them their name. So the threshold is 10 and the value is two and if I call this this way, this is also still false even though now I've called them out of order but by giving their names with what are called these keyword arguments, the interpreter is able to understand which variables I'm setting. But what I can't do, for example, is pass in a keyword argument like threshold as the first argument and then not provide a keyword for the second case that when we'll get a syntax error that says a positional argument follows a keyword argument. So any positional arguments that are going to be interpreted in the right order of the slots have to go first and then any keyword arguments come after those.

As I said it's often very useful to be able to use keyword arguments when there's a long set of arguments that can be possibly passed in and you might want to modify one of them halfway through the list. You don't have to type in all of the positional arguments up to the point where you get to reset one of those and we can see this, for example, in this Pandas module. Pandas is a very useful module we'll see more about and it provides a function called read_excel. When we look, we go through all the documentation for this function, we see that there are lots and lots of input arguments that we can possibly set and we don't want to have to type in all of those. For example, what if we just want to change this one input argument, this parse_dates. What we can do instead is that we can, for example, read an Excel spreadsheet and just pass in this one override, this one input argument parse_dates=True. So with that kind of functionality we can mix positional arguments and keyword arguments to be able to access a lot of different kinds of behavior in these different kinds of routines.

Back to Table of Contents

Module Wrap-up: Writing Custom Python Functions

You've examined the ins and outs of writing functions in Python, including how to define, write, and call a function. In addition, you explored how to use default and keyword arguments in functions.

Back to Table of Contents

Module Introduction: Using Basic Elements of Control Flow and Iteration in Python

In this module, you'll take a look at creating a data processing pipeline with three different types of control flow: sequential execution, looping and iteration, and branching. You'll also have the chance to practice different types of control flow. In addition, you will examine the importance of using indentation in your code as well as what happens if you misindent. Finally, Professor Myers will walk you through how to handle exception errors when you run your code.

Back to Table of Contents

Read: Creating a Data Processing Pipeline With Control Flow

☆ **Key Points**

Control flow involves three main types of data processing:

- sequential
- branching
- iteration

In an analogy to human language, data are the nouns of a data processing pipeline and functions (transformations) are the verbs. But much of human language hinges on other parts of speech, especially conjunctions like "and," "if," and "but"; these conjunctions unite words, phrases, and clauses. The conjunctions of a computer program are the elements of control flow, which give structure to a data processing pipeline in the form of loops, branches, and chains of related transformation. In this sense (to introduce a second analogy — see **Conjunction Junction**), a data processing pipeline is a bit like a network of train rails: The data (trains) are the objects that are shuttled about (transformed under their locomotion), but it is the switches and stoplights that represent the underlying intelligence of the operation. Python provides a number of syntactic constructs to guide the flow of data in complex programs.

Broadly speaking, control flow involves three main types of data processing:

- Sequential processing: a linear set of operations that transform data one step after another.
- Branching: choosing a particular path through a data processing pipeline based on conditions associated with the data.
- Iteration: repeating an operation (looping) over a set of instructions for a specified number of times or until some condition is reached.

Like most programming languages, Python provides language constructs for supporting these different modes.

Back to Table of Contents

Watch: Sequential Execution

Programming is largely about encoding data processing pipelines to shape the input data that we are given to work with into new forms of use to us. These data processing pipelines — using various operations to transform the current state of our data — can support various types of structural motifs, the simplest of which is sequential execution, where we break a more complicated operation into a sequence of simpler steps.

Video Transcript

So, when we're programming, what we're really doing is shaping data to fit our needs and we do this by encoding some sort of data processing pipeline, and we know that there are important strategies such as divide and conquer that let us break down complicated tasks into a series of simpler tasks, and this same strategy plays out in developing these data analysis pipelines, and so we can think of having data. This is the state of a program. We have different functions and other operations that can transform that data, and then we have different ways of imposing control flow on what that whole pipeline looks like, and again, the reason we need to do this is because we need to shape, it's like we're a blacksmith and we're taking the raw materials that had been mined out of the earth, and we're shaping it into some useful tool. If the data that you needed were already in the form that you needed to do your work, you would need to write programs to transform them.

So, in developing data analysis pipelines, we have a series of different kinds of operations. We have sequential execution, which is an important first step, in that we break down a complicated set of tasks into a series of simpler tasks, and so that the data returned by one operation becomes the substrate for whatever the next operation in the and the chain is, and we can build up these big chains to transform through a series of manageable steps into some form that we can act on.

Back to Table of Contents

Code: Calling Functions Sequentially to Process Data

Python statements are interpreted and execute one after another, so in some sense all of Python programming involves "sequential execution" (except for programs that run in parallel, at the same time on multiple processors, which is an advanced technique beyond the scope of this course). But it is often useful — as part of a "divide and conquer" strategy — to break up a stream of program statements into groups and assign intermediate results to variables that we can reuse later. We often choose to group a series of operations together and define a function that we can call to compute an output for a given set of inputs. With sequential execution, we're thinking specifically about a sequence of related operations that feed data through a computational pipeline.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Looping and Iteration

Looping and iteration are key elements of program control flow, allowing for code to act repeatedly on data in a streamlined fashion. `for` loops and `while` loops are two common mechanisms for iterating over data in Python.

Video Transcript

So when working with large datasets, it's often the case that many calculations are repetitive or need to be iterated many times. Unlike us humans, computers don't get bored with repetition and they're very fast at it, and they keep doing it even when we would have long since given up as humans trying to calculate. So, this looping structure, this iteration is often a central element of a data processing pipeline where we loop around through a series of objects, through a series of data, perhaps waiting until our calculation is finished.

So, there are two main iteration modes one finds in most programming languages. There is what's typically called a for loop, and this is where we loop through some specified number of times that we prescribed, or maybe loop through an object that contains multiple items that need to be accessed one by one. The other major iteration mode is a while loop. We loop until some condition is met, and then we exit. So, in Python we have support for both of these kinds of iteration modes.

The for loop is encoded with the for keyword, a for statement, and this allows us to loop over some kind of iterable, some kind of object that can be iterated over, and access each value in turn, and within the for loop, we indent within the body of the code block within the for loop, that body gets repeated over and over again such as in the simple example where we execute a series of three statements. So, we can iterate over lots of different types of things in Python, anything that can be iterated over for, we can write a for loop to traverse, and so for example, you might have a simple list containing the items one, 10, 100, and 1,000. So, we could say for element in this list and in successive turns to the loop, the element takes the values one, 10, 100, and 1,000. We could have a dictionary, for example, which contains a set of key-value pairs, and we could loop over the pairs of keys and values through a statement such as this, for key, value in adict.items, and this would then iterate through all of the key-value pairs in the dictionary and access them one at a time. Or we can loop over other objects, like the range object which produces a sequence of integers but is a special object that is able to do it almost magically. So, we could say for example, for i in range(0, n) and this will at each time through the loop i will acquire the value of zero, one, two, three, four up to n minus one, so going through the loop n times.

The while statement in Python executes this while loop, and this takes as an argument some Boolean condition. So, a Boolean is either true or false and the while loop will execute as long as that condition is satisfied, as long as it is true. So, we could say for example, while x is greater than zero. Then we do something within that loop over and over again until as long as x is greater than zero. When now if x is greater than zero forever, and it never changes, this program might never terminate. But hopefully you've implemented the right logic, and eventually that will stop when x becomes negative. We could loop over, we can ask for example while some element is in a set and as long as an element is in a set, we'd want to loop over and access that data, or we could for example, even say while true and then this truly would run on forever if we did nothing else, because true is always true. So, while true would run forever, but of course, we would typically have some other test within a loop like this that would allow us to break out of it.

So, the break statement is an important part of this control flow. So, with break, we can exit the loop immediately either within a for loop or on a while loop, and while some languages implement another construction, a do-while loop where something is done first and then the while is tested, the while condition is tested, we can basically emulate that in Python which does not have a separate do-while construct by using the break. So, for example, we could say while true, do something and then check a condition. If that condition is true, we might break from that loop.

So, these are the basics of how to loop through data and create iterations based upon the structure and value of that data.

Back to Table of Contents

Code: Iterating Over Lists and Dictionaries

Central to data science is the ability to operate on collections of data. This often involves writing `for` and `while` loops to iterate over the data. Sometimes we want to visit each element of a container and perform some operation on that element; `for` loops are useful for this. In other cases, we want to iterate until some condition is met; `while` loops are great for that. In both cases, the point is that you write the body of the loop once and let the loop execute that code multiple times with different pieces of data each time through. With this basic structure, you can process all the different data elements of a container or build up new data containers based on criteria that you specify.

In this exercise you will practice working with `for` and `while` loops. You will also be introduced to docstrings, or documentation string. Docstrings are an incredibly useful way of documenting what different functions, objects, classes and modules do in Python. More useful than mere comments that are ignored by the interpreter, documentation strings placed at the beginning of a function definition can be used to describe what the function does, and that documentation can be accessed through the built-in help functionality in Python and IPython.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Iteration Basics

Now it's time to test your knowledge about iteration.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Watch: Code Blocks: The Role of Indentation

Code blocks are chunks of code that are run sequentially within some context. For example, a code block in a `for` loop indicates what code gets repeated each time through the loop. Unlike many other programming languages, Python uses consistent indentation to delineate the code blocks used in function definitions, loops, and other elements of control flow. This means that all code indented at the same level is within the same code block. Other programming languages use punctuation (e.g., curly braces) or keywords (e.g., `endif`) to delineate blocks. Understanding how to indent code correctly and to recognize errors arising from misindentation are important skills in Python programming.

Video Transcript

So many language features in Python involve creating separate code blocks that get executed in certain contexts. We might, for example, have a function definition where we want to define what happens in the body of a function. We might have for loops or while loops where we're iterating through a code body, or we might, in an if statement, have a code block that only gets executed if a certain statement is true. So in Python, these code blocks get defined by their level of indentation. A spatial indentation in the code, and this is important to realize especially because this is somewhat different than is the case in many other programming languages.

So these code blocks, these indented code blocks, can often be nested within each other. So let's look at this little function here that we've defined called find_shared_items, and it takes two lists. What it's going to do is with those two lists, it's going to find all of the items that are shared in both list one and list two. We can do this, for example, by initializing an empty list shared_items and then looping through every element, every item in list one, looping through every item in list two, using the if statement to decide if item one is equal to item two and if so, appending that to our list of shared items, and then finally returning the set of shared items when we're all done. So in each of these nested code blocks, we have to indent over another set of spaces, and it's that indentation that defines what gets executed within each of those code blocks.

So other languages typically provide other sorts of additional syntactic elements to identify blocks. Many languages such as C, C++, Java, and Perl use curly braces to denote where the beginning or the end of a code block is, and then some other languages have additional keywords that are used to identify where the end of a block is. So in Fortran, there's the end do, and in the bash shell, there are keywords such as done and fi, which is the inverse of if. So what Python does is it has adopted a more clean, compact, and readable design philosophy using indentation to identify code blocks without having to rely on these additional sort of syntactic

elements. But it is important to remember that this is not just for readability but actually for functionality.

So both syntax errors and logical errors can result from inconsistent indentation, and so a typical sort of style guide recommendation is that we use spaces to indent into the new block and we use four spaces of indentation per block. It's important when you're creating Python source code of your own to use an editor that understands these rules of Python and why and how indentation should be done. So you want to use a Python-aware editor that assists with consistent indenting across multiple code blocks.

As a reminder:

- Both syntax errors and logical errors can result from inconsistent indentation.
- Style guide requires four spaces' indentation per block.
- It's important to use a Python-aware editor that assists with consistent indentation.

Back to Table of Contents

Watch: Branching

Branching is another central element of control flow in programs: executing different code based on the state of the data being processed. This is typically carried out using `if` statements, where one code path is followed if a condition is True, and another path otherwise. In this video, we'll see how Python supports this type of control flow.

Video Transcript

So, branching is another important structure found in data processing pipelines. This is where we branch, we take a fork in the road in our code based upon the state of the data typically determined by some Boolean condition by whether or not something is true or false. If keyword and associated elements such as if, else, and elif are all part of allowing us to implement this kind of branching functionality.

The basic structure of an if statement is we have if condition, some condition where it is, and then indented within that some code block that is executed if that condition is true. So, it can be as simple as we might check to see whether x is greater than zero. If x is greater than zero, then we execute something. But if x is not greater than zero, then we do not enter into that block and execute that code. We might check to see if a key is in a dictionary. So, if a key is not in a dictionary, we might, for example, want to first initialize a value for that entry in the dictionary, or you might say I want to initialize it as an empty list. But if the key already is in the dictionary, then we don't want to do that kind of initialization. We could even do something as simple as say if true, in which case we would do that always, but we might not readily encounter that in practice.

If comes along with associated other keywords such as else. So, we might want to do one thing if one condition is met and something else if that condition is not met. So, if x is greater than zero, we might do something; else, then we indent in a separate code block and we do something else if the opposite is true, if x is less than or equal to zero.

We can actually string together multiple if tests using the elif keyword. So, we might have, say, a variable level, and if the level is high, we want to do one particular thing. If the level is medium, we might want to do something else. If the level is low, we might need to do a third thing, and then at the end of that, we might, for some reason, there might be an error in our program, maybe the level got mistyped or there's some other entry that we don't know about. So, then we want to catch with else to do this last set of operations if none of those previous conditions are met.

So, with these if, elif, and else statements, we can execute complicated kind of branching through our data space.

Code: Writing If Statements

Branching is the process of following different paths through code depending on the current values of data elements. In Python, this is accomplished with `if` statements. An `if` statement tests whether a particular logical condition is True or False, then branches to one code block or another based on that condition. A logical test is often carried out with a relational (or comparison) operator that tests a relationship between two objects, such as whether they are equal or if one of them is greater than the other. Unlike arithmetic operators (such as addition and subtraction), which return numbers, relational operators return Boolean (bool) objects, either `True` or `False`. For example, the expression `a < b` will evaluate to `True` if `a` is less than `b` and `False` if it is not. These sorts of relational comparisons are used in `if` statements to branch to different blocks of code depending on the values of the data.

In this exercise, you will fill in some missing pieces of code in order to implement the correct branching.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Exception Handling

When you write code, you hope that things go as planned and that data are correctly processed. But sometimes things go awry: You might try to open a file that doesn't exist, or you might pass the wrong argument to a function such that the function cannot operate on that data. A number of different types of errors can creep into your code. Broadly speaking, there are three classes of errors (or "bugs") that can occur in a program: syntax errors, runtime errors, and logical errors.

Syntax errors occur when you violate the basic rules of the language, which the interpreter or compiler can detect and report when first processing the code. Runtime errors (or exceptions) arise when a syntactically correct piece of code is provided with data that cannot be executed; you might try to divide a number by 0, you might try to open a file that doesn't exist, or you might pass the wrong argument to a function such that the function cannot operate on that data. Logical errors involve code that is both syntactically correct and generates no runtime exceptions but which nonetheless computes the wrong answer due to an incorrect translation of an algorithm into computer code. These are the most insidious and difficult to track down.

As noted, syntax errors are detected and reported by the interpreter. Diagnosing logical errors requires careful validation of the results of your analyses. Exceptions are a class of errors that occur while a program is running, and Python defines many different types of exceptions to reflect the wide variety of different errors that can arise. A runtime error can often lead a program to crash, but Python provides, through the use of try-except blocks, the ability to catch these exceptions and deal with them usefully. Exception handling is a process by which you can write code to deal more gracefully with runtime exception of the sort discussed above.

Video Transcript

So, another important type of control flow in programs is exception handling. And Python provides some very nice tools for this although many programming languages do not really do so. So, exceptions are run-time errors, these are errors that are encountered while the program is running and so they're not syntax errors that can be identified ahead of time. So they arise during the execution of a program and are reported by the interpreter. An exception is raised when one of these errors is encountered and in fact there are many different types of exceptions that correspond to different types of errors that one can find in a program. If an exception is not caught and treated, a program can just crash and so it's important to be able to respond to these errors, but again, Python provides some nice capability in this regard.

So, let's consider a very simple example, let's say I want to divide one number by the other and I happen to divide, I want to assign this to variable x, and so x is equal to one divided by zero. But we know mathematically, we can't divide a number by zero and so what the Python interpreter actually throws and which is shown is a ZeroDivisionError. That this is the ZeroDivisionError is an object in Python that gets raised when we try dividing something by zero. There's another type of error say, a FileNotFoundError. If I want to read from a file, I can use the open keyword and so I might try to open some and give it some filename, but if that file does not exist what open will do is throw this FileNotFoundError. Again, if I don't do anything to catch these things, then my program will just terminate at that point.

But fortunately, exceptions can be caught and dealt with without having the program crash, and the keywords that are used to implement this are try and except. So, what we can do for example is we can try a calculation and within the indented code block, we could try for example assigning z to the ratio x divided by y, but if y happens to be zero and that calculation throws a ZeroDivisionError, we can actually catch that with the except keyword. So, we might for example, rather than having the program crash, we could print out some kind of warning message and we could reset z to some reasonable default based on the fact that we're dividing by some very small number.

Similarly, with reading a data file, we could catch this FileNotFoundError that would normally be thrown and we catch the file not found or we could print out some statement saying that the file cannot be found, asking the user to provide a different file name and we might return gracefully from the function rather than having the whole thing come to a halt. So, exception handling is an important part of control flow and as you use Python more and more you will sometimes see exceptions being thrown and it's useful to understand where they come from.

Back to Table of Contents

Watch: Abstracting the Process of Iteration

Part of the expressive power of the Python language is the broad applicability of particular operations across many different types of data. The Python container data types, as objects that hold other objects, all support iteration; that is, we can iterate through each of the elements one at a time. Python formalizes this notion with the concept of an "iterable" and abstracts the process of iteration to allow iteration over other types of objects as well.

Video Transcript

So, we've encountered the notion of an iterable which seems a little fuzzy, it's something that can be iterated over and this is an important notion in Python that part of what the language has done is abstracted the process of iteration because many different data types can be iterated over, the developers of Python wanted to make sure that the interface to that look common and then we can iterate through a list, we can iterate through a tuple, we can iterate through a dictionary. So, there's this important element, aspect of the abstracting the process of iteration and we'll look at a few examples of how this plays out in the interpreter.

So, with a for loop we can, for example, loop through a list explicitly, we can say for n in the list 0, 1, 2, 3 and print out n but we also can use the built-in range function to do the same kind of thing, for n in range(0, 4) print n and we get the same set of four values for n 0, 1, 2, and 3. But if we decide to ask what this range(0, 4) is, we see that it just returns a thing called range(0, 4) and if we ask for help on this, we see that what range does is it produces a range object. So, the range function produces this new object and the purpose of this object is to allow it to produce a sequence of integers through iteration. So even though the range object by itself doesn't seem very useful, as we saw we can iterate over it, it is an iterable that is able to produce a sequence of integers but it is not by itself the sequence of integers and this is what we mean by abstracting the process of iteration.

There's another function, built-in function called enumerate which is very useful that will take an iterable such as a list and will print and will throw each time through the list not just get the value of the list but also the position of that element in the list. So the zeroth element of the list is 10, the first element is 20, the second element is 30 and so on and so forth. But again, if we look at what enumerate returns, we see that it returns some enumerate object and in fact if we ask what the type of that object is, we find out that it's of type enumerate. It's another type of a thing that can be iterated over and produce a sequence of data but without being the sequence of data itself.

Then finally, there's another function known as zip which will zip two lists together, for example, two iterables together. But again, zip creates a zip object which can be looped over rather than zipping together two functions themselves. So, these are all examples of what are known as lazy evaluation that rather than require us to build up explicitly, say a list of integers that might be very large, we can produce values one at a time as needed, this is the sense in which the range, for example, is lazy that it only it can produce an integer one at a time but it only does it as asked for.

So this in fact uses less memory because we don't have to create a big list, it's often faster because it can keep track of which ones come next in the list. We can always create an explicit list if we desired. We could, for example, say list(range(100)) and get out that list of integers but we don't have to. It's also important to note that while we've been talking mostly about sequences, about ordered sequences that we can iterate over such as lists and tuples, dictionaries and sets are also iterable but their iteration order is not guaranteed. The order of data in a dictionary or a set has no meaning. So although you can iterate over it and access all that data, you won't necessarily be guaranteed to get them in any particular order.

Back to Table of Contents

Code: Investigating Iterables

In various places, the Python documentation makes reference to "iterables." As the name implies, iterables are objects that can be iterated over; in other words, containers that can be accessed so that each element is visited once and only once. Lists, tuples, sets, strings, and dictionaries are all specific examples of iterables, but the concept is more generic. There are a variety of operations that can be performed over all these different types of objects, including:

- Iteration over the elements of the container in a `for` loop.
- Use of the function `len` to determine how many items are in the container.
- Use of the `in` keyword to determine if a particular item is in the container.

We can use the operations listed above with any iterable. Some other operations, however, only apply to specific containers, such as appending to the end of a list (which we can't do with a set or a dictionary, since the "end" has no meaning). It turns out that other, more specialized objects produced by some of Python's built-in functions are also iterables, even if they don't appear to be when you first look at them. This includes objects returned by the functions `range`, `enumerate`, and `zip`.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Read: A Review of Indexing and Comprehensions

Since we're reviewing Python iterables, it would be helpful to review containers a bit, as well as introduce you to Comprehensions. Note that while you are not expected to fully understand Comprehensions at this time, it is worth reviewing them, as they will appear in a few examples in this course.

Note: Please review both the Indexing and Slicing tab and the Comprehensions tab below.

- **Indexing and Slicing**
- **Comprehensions**

Indexing and Slicing

Lists

Elements in Python lists can be *indexed* by an integer position in the list, with the index starting at 0 and ending at n-1, where n is the number of elements in the list. For example,

- `my_list[0]` refers to the first element in `my_list`.
- `my_list[1]` refers to the second element in `my_list`.
- `my_list[-1]` refers to the last element in `my_list`.

That last example might seem surprising, but Python interprets negative index positions as counting backward from the end of the list. So, for example,

- `my_list[-2]` refers to the next-to-last element in `my_list`.

We can use indexing to either *get* the current value of a list at the specified position or *set* the value at that position to something new; e.g.,

- `my_list[-2] = 12`

Python extends the notion of indexing to *slicing*; i.e., grabbing contiguous chunks of a list by specifying both a start index and a stop index, separated by a colon `:`. The slice ends one position short of the stop index so that the total number of elements in the sublist that is sliced out is equal to `stop - start`. Slicing syntax looks like this:

- `my_list[0:3]` refers to the first, second, and third elements of `my_list`.
- `my_list[:3]` also refers to the first, second, and third elements of `my_list` — if the start index is omitted, it is assumed to be 0.
- `my_list[3:6]` refers to the fourth, fifth, and sixth elements of `my_list`

- `my_list[3:]` refers to all the elements of the list, starting with the fourth and continuing through the end of the `my_list` — if the end index is omitted, it is assumed to continue to the end of the list.

Strings and Tuples

Because strings and tuples also represent ordered sequences of data, the same indexing and slicing syntax to access data also applies to strings and tuples. But because strings and tuples cannot be modified after they have been created, we cannot use indexing and slicing to set their data to new values.

Dictionaries

Elements in Python dictionaries can be *indexed* by a key, using the same square bracket syntax as for lists. And because dictionaries can be modified, the same indexing syntax can be used to add new key-value pairs or change the value associated with a key.

Because the order in which key-value pairs appears in a dictionary has no meaning, it does not make sense to slice out multiple items from a dictionary.

Sets

Because a set is an unordered collection of unique items, neither indexing nor slicing a set are defined operations. One could, however, convert a set to a list, then index and slice that new container.

Comprehensions

Comprehensions are a group of expressions that create different types of containers with a compact syntax:

- List comprehensions create lists (indicated by enclosing the expression in square brackets).
- Dictionary comprehensions create dictionaries (indicated by enclosing the expression in curly brackets, with key-value pairs separated by a colon).
- Set comprehensions create sets (indicated by enclosing the expression in curly brackets).

They all have an analogous structure, although with some differences among them. Consider the generic syntax for a list comprehension:

```
[expression for element in iterable if condition]
```

Let's break this down in more detail:

- `expression` indicates what will be inserted into each element of the list.

- `for element in iterable` indicates that some iterable object will be iterated over (e.g., a list) and that the variable `element` will, in turn, acquire the value of each successive item in the iterable.
- `if condition` indicates that `expression` will be inserted in the resulting list if the `condition` is True.

We've already looked at this example:

```
squares_even = [n**2 for n in range(100) if n%2==0]
```

A verbal way of summarizing what is produced by this list comprehension is: "Produce a list of numbers consisting of the squares of all the integers starting at 0 and proceeding up through 99 if that integer is even; i.e., if the remainder (or mod) resulting from dividing that integer by 2 is equal to 0."

If no condition expression is included at the end of the comprehension statement, then the resulting list will have as many elements as the original iterable. The most boring list comprehension of all is that which simply copies all the elements of a list; i.e.,

```
[e for e in my_list]
```

Set comprehensions create sets using the same syntax as for list comprehensions, except for enclosing the expression in curly brackets rather than square brackets. In addition, because sets can only contain unique elements, if there are duplicates, only one copy is included in the result:

```
{expression for element in iterable if condition}
```

Dictionary comprehensions can be identified by the fact that the `expression` that is inserted into each entry in the dictionary is not a single element but a key-value pair (separated by a colon).

```
{key_expression: value_expression for element in iterable if condition}
```

For example, consider this expression:

```
dict_squares = {n: n**2 for n in range(100)}
```

A verbal way of summarizing what is produced by this dictionary comprehension is: "Produce a dictionary whose keys are the integers starting at 0 and proceeding up through 99, and whose values are the squares of each of those integer keys."

While in these examples we've been using relatively simple expressions to be inserted into the containers, any arbitrarily complicated expression that returns a value can be used in the first slot in a comprehension statement. But if you need to build up a list based on the values of a very

complicated expression, you might be better off using a `for` loop and building up incrementally. The power and expressiveness of comprehensions is alluring, as is the satisfaction of bundling up a complicated computation in one line, but there is no shame in unrolling that and using a multiline `for` loop if that works better for a particular problem.

Back to Table of Contents

Read: Putting it All Together: Building Data Processing Pipelines

> ☆ **Key Points**
>
> When building a data processing pipeline, consider program state, transformations, and control flow.

Writing a program is essentially the process of constructing a data analysis pipeline. Even though we might think of such a pipeline visually or with a metaphor such as trains, rails, and switches in a rail yard, what we are really doing is using the elements of a programming language to hook all those pieces together. In building a data processing pipeline, you need to think about:

- Program state: What data do I need to move through my pipeline and what do I need to transform it to in order to get the type of output in which I am interested?
- Transformations: What functions do I need to transform, modify, combine, or filter the data that I am moving through the pipeline?
- Control flow: What sequence of transformations do I need to hook together, and how does the path through the pipeline depend on the particular values of the data that are moving through it?

Back to Table of Contents

Module Wrap-up: Using Basic Elements of Control Flow and Iteration in Python

You now understand the different types of control flow and how to use them when creating a data processing pipeline. You looked at how indentation is used to define blocks of code to be executed, and you examined exceptions that get generated in your code and how to handle them.

Back to Table of Contents

Module Introduction: Creating New Custom Python Data Types With Classes

In this module, you will examine object-oriented programming and custom data types. You'll explore how to use classes to bundle data and methods to create new data types. You will also have the chance to practice building a custom class and creating new objects.

Back to Table of Contents

Read: Object-Oriented Programming and Custom Data Types

Object-oriented programming (often abbreviated as OOP) describes a style of programming and a class of programming languages. But since there are many aspects that help to define what OOP actually is, different languages support, require, or embrace OOP to different degrees. Although Python is at its core an object-oriented language, it mixes support for OOP along with support for other programming styles, such as procedural programming and functional programming, allowing programmers the flexibility to use whatever approaches they prefer or find best suited to a problem. Other OOP languages can be more dogmatic in their approach.

☆ **Key Points**

Object-oriented programming typically involves:

- Calling methods on objects to activate some desired functionality.
- Providing support for defining new custom data types (classes) with associated functions (methods).

Everything in Python is an object.

Python provides excellent support for defining new classes of objects.

OOP typically involves:

- Calling methods on objects to activate some desired functionality.
- Providing support for defining new custom data types (classes) with associated functions (methods).

Everything is an object

We have noted that everything in Python is an object, with an associated namespace. We can list the names of the items in that namespace with the built-in function `dir()`, and we can see documentation about what is defined in that namespace with the function `help()`.

Even when we call a built-in function or operation, such as `len` or `+`, what we are really doing is calling a method on the object that is passed to one of those functions. For example, both a list and a dict have a method named `__len__`, and when we compute the length of a list or dict using the built-in function `len`, what is really called underneath is the `__len__` method associated with that object. As an example, let `my_list` be a list:

```
my_list = [1,2,3]
len(my_list)  # returns 3
my_list.__len__()   # also returns 3
```

Even more basically, when we add two integers using the `+` operator, what we are really doing is calling the `__add__` method on an integer object:

```
2 + 3 # returns 5
(2).__add__(3) # also returns 5
```

The fact that the same `+` operator can be used to concatenate two strings is facilitated by the fact that a string object defines its own version of `__add__`:

```
'abc'.__add__('def') # returns 'abcdef'
```

Support for new objects

Python also provides excellent support for defining new classes of objects, and the Python data science ecosystem has developed a number of new classes to support complicated workflows and data analysis pipelines.

Often, new objects are needed to bundle together other data types and functions in order to capture new concepts. Those concepts are sometimes called *abstractions* that identify a new entity that is somehow more than just the sum of its parts. The built-in data types provided by Python are a powerful set of primitives which can be mixed and matched in different combinations to produce more complicated data types with their own behavior. The ability to define new classes provides a convenient means to bundle related data elements together in one object rather than having to keep track of several different objects, but more importantly, it also provides a way to define new primitives. In the same way that we use a divide-and-conquer strategy to break down a complicated function into an interacting set of simpler functions, we can build complex data types out of interacting hierarchies of simpler data types.

Back to Table of Contents

Watch: Bundling Data and Methods Using Classes to Create New Data Types

As an object-oriented language, Python provides the ability for you to define new types of data, useful for capturing concepts and abstractions in a particular application domain or business setting. In this course, you will mostly be using custom data types defined in different packages rather than creating new data types yourself. In either case, however, it is useful to have a sense of what Python classes are and how they operate.

Video Transcript

So, just as we can define new functions in Python, we can also define new data types or as we call them classes because essentially what we're doing is we're describing classes of related types of objects. So, we use the class keyword in Python to define new data types and then we create instances of those objects of that class, which are our objects that we manipulate in our program. These new data types are often very useful for bundling together related sorts of data as well as being able to attach different kinds of functions and methods tied to that data, that are able to act on that data.

You can think a little bit of, for example, just the built-in Python dictionaries, that they are themselves rather complex objects that tie together keys and values and provide various methods to access and manipulate those data, but we can in fact use the class keyword to define new data types that don't exist in the Python universe. So, we're going to walk through an example. Shortly, I'm defining a new data type for pizzas that Python doesn't have a built-in pizza datatype, but maybe we run a pizza parlor, and we want to keep track of the pizzas we make and how much they cost. So, we can bundle together data describing the size of the pizza and the toppings on a pizza and methods for calculating the price, and we use this using a combination of these various keywords.

We use the class keyword to define a new data type, we use the def keyword, which we've already seen used to define standalone functions, but we use that same def keyword to define methods attached to this data type, and then we'll see that there's kind of a funny new wrinkle, there's a self parameter, that is used to attach function definitions, to attach the class of all pizzas to particular pizzas that we want to build. The self keyword is a little bit like the this keyword in C++ and Java if you've seen that, but it behaves a little bit differently as well.

So, let's go on in the interpreter and see what this looks like. So, we've got some code here that defines a class, a new data type called pizza, and as noted, we use the class keyword to let the interpreter know that we're defining a new data type and then we have within this class

definition two method definitions, two functions that act within the universe of pizza pies. So, we have an initialization method that lets us build a pizza of the specified size and with the specified set of toppings, and we have a price method, that calculates the price of the pizza based upon the size and the number of toppings.

So, what we can do for example is we can use this initialization method to create a new pizza. We're going to create a large pizza with garlic and pepperoni and that's my pizza and then I'm going to create a pizza for you, you don't want garlic and pepperoni, you want instead onions, peppers, and olives. So, what's happening here when I'm calling this function pizza, what I'm really doing is I'm calling this initialization method that I've defined, that takes an argument, which is the size, and it takes an argument, which is a set of toppings. But, you see when I define this thing, I actually had a third argument itself in the function definition but when I call pizza, there's only two arguments, there's a size and there's the set of toppings. What the self is essentially doing is it's tying. Pizza is creating a specific pizza in the first case, that's specific pizza is my pizza, and in the second case, the specific pizza is your pizza and the self is essentially referring internally to which pizza we're talking about. So, it's something that it's a keyword that you use in defining a function but not in calling the function.

But now, that I've created these things, I can ask for example what is the type of each of these objects? So, my pizza is a pizza object, I can ask for example what is the price of my pizza based upon its size and toppings? That's $15, and I can do the same thing with your pizza, that's 1350 because it's a smaller pie. So, by using the class and def keywords, we can build new complicated data types, that bundle different elements together and represent their relationships.

Just as we saw, we could use docstrings to document functions, we also can use docstrings to document both classes themselves and methods within those classes. So, for example, I can do a help query on pizza, I can do a help query on pizza price and in both cases I get the docstring associated with either the class definition or the method. So, this is a very useful tool both for you writing your own new custom objects as well as when you encounter different packages or written Python often those are themselves creating new data types based upon the class keyword.

Back to Table of Contents

Code: Building a Custom Class and Creating New Objects

As an object-oriented programming language, Python supports the creation of custom data types, enabling you to capture relationships among data elements and define specialized methods on those data types. This is sometimes useful when multiple data items are used together to carry out a computation — they can be bundled into a single object, with a defined set of operations that reflect the new type of object being created. The `class` keyword enables you to define a new class in Python, just as the `def` keyword enables you to define a new function. When just getting started, you are more likely to be using custom data types developed by others rather than creating your own, but it is useful to understand that you have this power, just as you have the power to define new functions to carry out custom tasks. We will explore in more detail how class definitions work by constructing the Pizza class that Professor Myers introduced in the previous video.

This activity will not be graded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Defining Classes, Including the Role of the 'self' Argument

Now it's time to test your knowledge of defining classes.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve 100%.

Please complete this activity in the course.

Back to Table of Contents

Read: Custom Data Types Provided by Other Python Libraries

As we have seen, Python also provides excellent support for defining new classes of objects, and you've gotten some experience with doing that yourself.

> ☆ **Key Points**
>
> There are lots of new classes of objects that have been created by other developers using Python in order to capture important data relationships.

At the same time, the Python data science ecosystem has developed a number of new classes to support complicated workflows and data analysis pipelines. As you dig deeper into the Python data science ecosystem, you will encounter all sorts of new classes of objects that have been created by other developers using Python in order to capture important data relationships that are applicable to a variety of problems. This will include objects such as arrays and DataFrames for working with numerical and tabular data, figure and axes objects for making plots and charts of data, and estimators for making predictions about data using machine learning.

The Python development community appreciates the value of creating a "Pythonic" solution to a problem at hand and building on the base elements of the core Python language to make intuitive the operation of new data types produced by external libraries. So the time you have spent learning about indexing and slicing with built-in Python lists, for example, will be very valuable when you work with other data containers such as arrays and DataFrames. The fact that you know how to interrogate an object to figure out what sorts of methods it defines will enable you to learn how to use a new package to carry out your work.

Back to Table of Contents

Module Wrap-up: Creating New Custom Python Data Types With Classes

You have now explored how to use classes to bundle data and methods to create new data types. You practiced building a custom class and creating new objects, and you discussed what new data types would be useful to assist with problems on which you are interested in working.

Back to Table of Contents

Module Introduction: Crafting Numerical Calculations in Python

In this module, you'll examine how you can combine Python's functions and data types to create custom calculations. You'll apply this to the big picture of constructing data processing pipelines and practice using compound interest calculations.

Back to Table of Contents

Watch: Combining Python's Functions and Data Types for Custom Calculations

You've been learning a lot about the various pieces of the Python language and the mechanics of how they work. Ultimately, you want to use that knowledge to craft custom calculations of interest, using programming to go beyond canned calculations that might be provided for you. In this video, we'll consider a simple yet representative example of such a calculation.

Video Transcript

So, let's see how we can bring all the different pieces together. We've looked at numerical data types, we've looked at built-in containers within Python, and we've looked at the ability to define new custom functions, and what we really want to do is to figure out how to combine these different elements to do some calculation of interest. So, let's look at an example, say, of revenues for a business that we're running and we'll do this in the interpreter.

So, let's say, we've got our data that we've been collecting on all of our monthly expenses and revenues, we store these in lists called expenses and revenues, and they show up down below in the variable explorer. These are both length 12 because we've got a number for each month over the course of the year, and we can, for example, very easily calculate our monthly gross revenue. It's just the difference between the revenues and the expenses with a list comprehension. That in a single line here, we can calculate the difference between, within each month, the revenues and the expenses.

We might be interested in various kinds of summary calculations associated with this. So, for example, we might be interested in the mean value, say, over part of the year, and so we can define a function, say, for calculating the mean of something of a list entered. So, the way we calculate the mean of a list, for example, is by summing up all the elements and dividing by the length of the list, and we've embedded this in a try-except clause, just to make sure that if we don't pass in a list, things don't just crash out. So, we could, for example, say, what is the mean of all of our expenses over the year? What is the mean of all of our revenues over the year? This is accomplished just by returning this ratio of the sum to the length for each of the different lists.

We can even, using slicing, if we wanted to say, for example, let's say, I don't want to know just about over the course of the year, I want to know just for the first quarter, say, the first three months of the year, I can have what were my expenses? I can slice out just the first three elements of that expense list and get this. So, we can combine all these things, for example, to compute quarterly summaries of our expenses and revenues, and there's a big function that I've defined here, compute_quarterly_figures, this takes the list of expenses and revenues, and while

we're not going to go through this right here in gory detail, essentially, what this does is, it will loop over four quarters of the year 1, 2, 3, 4. Find the start and the end months for that segment. So, for example, the first quarter would be month 0, 1, and 2, as shown in this slice here, and then we can calculate the total expenses, the mean expenses, the gross revenues, and print out a summary, all within this relatively simple loop.

So, for example, if we now execute this code, what we get is a set of quarterly data, where with this print statement, I've printed out a summary for each quarter, Q1, 2, 3, and 4. What the mean revenues in that quarter, what mean expenses, and the gross, and by combining our built-in data types that allow us to hold collections of data by defining custom functions for things that we need to be able to compute, we can very easily craft custom calculations. We write these in source code, but now, when we get next year's data, we can easily re-run the code on next year's expense and revenue data, and this is the kind of thing that we do within Python on a regular basis, building up custom calculations for questions of interest.

Take a moment to review the code Professor Myers presented in this video.

```
# monthly expenses and revenues over course of year
expenses = [40000., 42500., 47000., 40500., 38000., 36000., 39500., 43000., 47000., 48500., 49000., 5000
0.]
revenues = [38000., 42500., 52000., 50000., 48000., 43000., 44500., 45000., 50000., 49500., 52000., 5400
0.]
monthly_gross = [revenues[i]-expenses[i] for i in range(12)]

def mean(x):
    """
    return the mean value of the elements in x
    by summing the elements and dividing by the length;
    if the input type does not support this,
    return the input itself
    """
    try:
        return sum(x)/len(x)
    except TypeError:
        return x

def compute_quarterly_figures(expenses, revenues):
    quarterlies = {}
    for quarter in [1,2,3,4]:
        start = (quarter - 1)*3
        end = start + 3
        total_expenses_qtr = sum(expenses[start:end])
        mean_expenses = mean(expenses[start:end])
        total_revenues_qtr = sum(revenues[start:end])
        mean_revenues = mean(revenues[start:end])
        total_gross_qtr = total_revenues_qtr - total_expenses_qtr
        summary = 'Q{}: rev = {:.2f}, exp = {:.2f}, gross = {:.2f}'
        summary_line = summary.format(
            quarter, mean_revenues, mean_expenses, total_gross_qtr)

        print(summary_line)
```

```
        quarterlies['Q{0}'.format(quarter)] = \
          (mean_revenues,mean_expenses, total_gross_qtr)
    return quarterlies
```

Back to Table of Contents

Read: Putting it All Together: Constructing Data Processing Pipelines

You've been learning about a lot of the basics of Python:

- Built-in numerical data types
- Mathematical operations for numerical data types
- Containers for storing groups of related data
- Syntax for defining custom, reusable functions
- Language features to control the flow of data processing pipelines

> ☆ **Key Points**
>
> Next, you'll complete an activity to construct a useful data processing pipeline.
>
> The design of a pipeline needs to take place from the top down.

That is a lot to keep track of, and when you're learning a new language, it's sometimes hard to see how all the pieces fit together. In the next activity, you'll get some experience combining these different pieces to construct a useful data processing pipeline. While all the nuts and bolts that you've been learning about are key to building such pipelines, the design of a pipeline needs to take place from the top down. You need to ask yourself:

- What do I need to produce?
- What are the inputs required to get the answer I need?
- What computations and components do I need to get to the end result?

Once you've worked out those high-level requirements, you then dig into your Python bag of tricks and ask how you can best solve the problem.

Back to Table of Contents

Assignment: Build Custom Data Processing Pipelines

Functions, data types, containers, and control flow are all tools that you use to build up custom data processing pipelines. Being able to combine these elements to carry out computations is key to the process of doing data science. Now that you've learned how to write functions and use containers in Python, you can start to build up more complicated sorts of calculations.

In this assignment, you write some code to compute the effects of compound interest; that is, the change over time of an account balance that increases at a particular interest rate per period.

This assignment will be graded.

When you finish your work:

1. Submit your work by clicking **Education —> Mark as Completed** in the upper left menu

2. This assignment will be auto-graded and can be resubmitted. After submission, the unit will remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education —> Mark as Uncompleted,** then proceed to make edits. Once you are ready to resubmit, follow steps one and two.

This assignment will be auto-graded.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Crafting Numerical Calculations in Python

You've now examined writing custom Python functions, classes, and workflows. In this module, you put it all together and looked at how you can combine Python's functions and data types to create custom calculations.