Code With Ducks

Johan Nguyen

This curriculum introduces high school students to foundational skills in **Python programming**, **data analysis**, and **mathematical concepts** such as **Algebra 2** and **Statistics**, while fostering **critical research** and **soft skills** for university success. With 10 structured sessions combining theory and hands-on activities, the program equips students to navigate academic challenges, analyze data effectively, and apply computational thinking to real-world problems. Designed for accessibility and engagement, it aims to inspire and prepare students for future research opportunities in STEM fields.

January 19, 2024

Contents

1. How to Read and Understand a Research Article (Step-by-Step Guide)

Reading a research paper can feel overwhelming, especially with technical terms and complex ideas. To make it easier, break it down into three steps:

1. Skim
2. Read Deeply
3. Analyze & Ask Questions

1.1. Step 1: Skim the Paper (Get the Big Picture First)

Before reading every word, quickly skim the paper to understand its main points. Focus on the following sections:

1.1 Read the Title & Abstract

- Title: Tells you what the paper is about.
- Abstract: A summary of the study, including the research question, methods, key findings, and conclusion.

Example:

- If a paper is titled "The Impact of Sleep on Student Performance", expect it to discuss sleep patterns and grades.
- If the abstract says, "This study found that students who sleep 7-9 hours perform better on tests," you already know the main result.

1.2 Look at the Headings & Figures

- Scan section titles (Introduction, Methods, Results, Discussion) to see how the paper is structured.
- Look at graphs, tables, and charts—they summarize key findings without requiring deep reading.

Example: A bar graph showing "Average Test Scores vs. Hours of Sleep" might reveal that 6-8 hours of sleep leads to the best performance.

1.3 Read the Conclusion

- This tells you the final takeaway.
- If the paper is relevant to what you need, then go back and read it deeply.

Example: If the conclusion says, "Our findings suggest students should prioritize sleep to improve academic performance," then you know the paper supports that idea.

1.2. Step 2: Read Deeply (Break It Down & Take Notes)

Once you decide the paper is worth reading, go through it carefully. Focus on:

2.1 Introduction: Find the Research Question

- Read the first few paragraphs carefully.
- Identify the problem the study is trying to solve.

- Highlight unfamiliar words, look them up, and write down their definitions.

Example: If a paper states: "This study examines how different sleep durations affect memory retention in high school students,"
- The research question is: Does sleep affect memory retention in students?

2.2 Methods: Understand How the Study Was Done

- Look for the number of participants (sample size), how data was collected, and any experiments performed.
- A small sample size (e.g., 10 students) may not be reliable.
- A large sample size (e.g., 1,000 students) is more trustworthy.

Example: If the method says, "We studied 500 students by tracking their sleep hours and giving them memory tests,"
- This means the study observed real students and tested their memory, making it more credible.

2.3 Results: Identify Key Findings

- Read tables, graphs, and numerical results.
- If there are percentages or statistical values, focus on what they mean.

Example: "Students who slept 7-9 hours scored 15% higher on memory tests than those who slept 4-5 hours."
- Key finding: More sleep leads to better memory performance.

2.4 Discussion: Connect Findings to the Big Picture

- The authors explain what the results mean.
- Look for phrases like "Our findings suggest…"
- Also, check for limitations, because every study has flaws.

Example: The paper might say, "One limitation is that we only studied students from one school, so results may not apply to all students."
- This tells you the study might not be universally accurate.

1.3. Step 3: Analyze & Ask Questions (Think Critically)

Once you understand the paper, go a step further.

3.1 Check for Bias & Credibility

- Who funded the research? If a soda company funds a study about sugary drinks, it might be biased.
- Is the paper peer-reviewed (published in a reputable journal)?

Example: A study titled "Energy Drinks Improve Focus" funded by an energy drink company might be suspicious.

3.2 Compare with Other Studies

- One paper isn't enough. Look for other research on the same topic.
- If multiple papers say the same thing, the finding is more reliable.

Example:
- If five different studies show that more sleep improves memory, it's strong evidence.
- If two studies say the opposite, you need to dig deeper.

3.3 Summarize in Your Own Words

- Try explaining the study to a friend or writing a 2-3 sentence summary.
- If you can explain it simply, you truly understand it.

Example: "This study shows that students who sleep 7-9 hours perform better on memory tests. The study used 500 students, making it fairly reliable, but it only studied one school, which is a limitation."

1.4. Tools to Make Research Easier

1. Where to Find Reliable Papers

- Google Scholar → scholar.google.com (Find academic papers)
- PubMed → pubmed.ncbi.nlm.nih.gov (Medical research)
- arXiv → arxiv.org (Computer science, physics, AI papers)

2. Understanding Difficult Terms

- Google → Type "define [word] in simple terms" to get a quick explanation.
- ChatGPT → Ask for simple explanations of complex topics.
- Wikipedia → Good for general understanding (but always verify information).

3. Verifying Sources

- Cross-check information: Search for multiple studies that say the same thing.
- Look at author credentials: Are they experts in the field?
- Check the journal: Is it well-known and peer-reviewed?

1.5. Example Exercise

Scenario: You're given a research article titled "The Effect of Blue Light on Sleep Patterns." Link: https://pmc.ncbi.nlm.nih.gov/articles/PMC9424753/°

Task:
1. Skim the title, abstract, and conclusion—what is the main idea?
2. Read Deeply and write down:

- Research question
- Key findings
- Any limitations

3. Analyze & Ask Questions

- Who funded the study?
- Do other studies support this finding?

2. What exactly is Programming?

Short answer: The collaboration between humans and computers

Long answer: Programming involves giving a computer a set of instructions to execute.

Example: Imagine your friend is reading a recipe on how to cook a crocodile from Gordon Ramsay.

In this scenario, Gordon Ramsay is like the programmer, providing instructions. Your friend, acting like the computer, reads and follows these instructions. The complexity of the recipe affects the complexity of the dish, just as in programming, the complexity of the instructions influences the complexity of the outcomes!

Programming is all around us. Whether it's ordering food online, watching a live stream on Twitch, or using self-driving cars, programming plays a crucial role in making these technologies work.

2.0.1. Human-Computer Collaboration

Computers excel at tasks that are repetitive and require high accuracy, something that can be challenging for humans.

Example: Computers don't need to eat or rest, allowing them to work continuously without fatigue. They also don't get bored or lose focus, which is essential when performing monotonous tasks.

On the other hand, humans are great at creative thinking, emotional understanding, and adapting to new situations—areas where computers typically struggle. Humans can also travel and experience the world, adding context and depth to their decisions, something a computer cannot do. This complementarity between human abilities and computer efficiency is what makes the partnership between humans and technology so powerful.

Human	**Computers**
Inconsistent	Consistent
Creative	No creativity
Slow and sometimes inaccurate	Fast and accurate

When we write code, we're essentially communicating with a computer. However, because computers operate differently from humans, we need to translate our instructions into a language that computers can understand.

In this course, we'll focus on Python, one of the most popular and widely used programming languages. Python is known for its simplicity and versatility, making it a great choice for beginners and experienced programmers alike.

2.0.2. PROGRAMMING IS FOR EVERYONE

Learning to code is an exciting adventure. Whether you want to do academic research, create apps, develop games, or build robots, coding is a valuable skill that can open up many opportunities. Remember, computers are tools designed to help us solve problems and create amazing things. While programming might seem challenging at first, sticking with it can lead to incredible achievements. Keep at it, and you'll be amazed at what you can create!

2.1. Python Fundamentals

2.1.1. Program

We communicate with computers by writing our commands in a text file using a programming language. These text files are known as programs. When we run a program, we are essentially instructing the computer to read the file, translate the written commands into operations it understands, and then carry out those operations. This process transforms our ideas into actions that the outer executes.

```
program_name = "Code With Ducks"
print("Welcome everyone to " + program_name + "!")

Output: Welcome everyone to Code With Ducks!
```

2.1.2. Print

Now what we're going to do is teach our computer to communicate. In Python, `the print()` function is used to tell a computer to talk. The message to be printed should be surrounded by quotes:

```
print("I have a dog") # Output: I have a dog
```

In the example, we direct our program to `print()` a compliment. The printed words that appear as a result of the `print()` function are referred to as output.

2.1.3. Strings

Computer programmers call blocks of text `"strings"`. In our previous example, we created the string "I have a dog." In Python, a string can be enclosed by either double quotes `("Hello")` or single quotes `('Hello')`. It doesn't matter which type you use, but it's important to stay consistent to keep your code easy to read.

2.1.4. Variables

Programming languages allow us to store data for later use. For example, if we need to remember a name, reuse a date, or keep track of a user ID, we can use variables to hold these values. In Python, we create a variable by assigning it a value with the equals sign `(=)`.

```
my_age = "I am 18 years old"
print(my_age)

Output: I am 18 years old
```

In this example, we stored the sentence "I am 18 years old" in a variable called `my_age`. Variable names can't contain spaces or symbols except for underscores `(_)`, and they can't start with numbers, though they can include numbers after the first letter (e.g. `a_variable_5` is valid).

```
message_string = "Hello"
print(message_string)

Output: Hello

message_string = "Goodbye"
print(message_string)

Output: Goodbye
```

Here, we first create the variable `message_string` and assign a greeting message to it. After printing the greeting, we change the `message_string` to a farewell message and print that, demonstrating how the content of a variable can vary while the process remains consistent.

2.1.5. Errors

Humans are prone to making mistakes, and since humans are the ones who write computer programs, errors are inevitable. To help with this, programming languages are designed to identify and explain mistakes in code.

In Python, these mistakes are called errors, and the language will indicate where an error occurred using a caret `(^)` symbol. When unexpected errors occur during program execution, we refer to these as bugs. The process of updating the program to eliminate these unexpected errors is known as debugging.

Two common errors you might encounter while writing Python are `SyntaxError` and `NameError`:

A `SyntaxError` indicates a problem with the structure of your program, such as misplaced punctuation, an unexpected command, or a missing parenthesis.

A `NameError` occurs when Python encounters a word it does not recognize, typically because it looks like a variable name that hasn't been defined yet.

Understanding these errors is crucial for troubleshooting and improving your coding skills.

```
message_string = "Hello'
print(message_string)

Output: SyntaxError: unterminated string literal (detected at line 1)

print(message_str)

Output: NameError: name 'message_str' is not defined
```

2.1.6. Numbers

Computers can process more than just text; they can also handle various numeric data types. Python, for instance, offers several ways to store numbers, depending on what you need them for.

An integer, or int, is a whole number without a decimal point. It includes all positive and negative counting numbers, as well as zero. For example, if you were counting the number of people in a room, the number of jellybeans in a jar, or the number of keys on a keyboard, you would likely use an integer.

A floating-point number, or float, is a number with a decimal point. It's used for more precise measurements or to represent fractions. If you were measuring the length of your bed, calculating the average test score of a class, or recording a basketball player's 3-point average, you'd probably use a float.

Here's how numbers can be assigned to variables or used directly in a program:

```
an_int = 2
a_float = 2.1
another_int = 3
print(another_int + 3)

Output: 6
```

2.1.7. Calculations

Computers are exceptionally good at performing calculations. Python can handle basic arithmetic operations like addition, subtraction, multiplication, and division using the symbols +, -, *, and /, respectively.

Here are some examples:

```
print(232 - 54 + 2) # Output: 180

print(10 * 2) # Output: 20

print(10 / 5) # Output: 2.0
```

Notice that when we perform division, the result includes a decimal point. This happens because Python automatically converts integers `(ints)` to floating-point numbers `(floats)` before dividing.

There's also a special error associated with division: `ZeroDivisionError`. Python will raise this error if you try to divide by 0, which is mathematically undefined.

```
print(10 / 0)

Output: ZeroDivisionError: division by zero
```

2.1.8. Changing Numbers

Variables that hold numeric values can be used in calculations just like numbers. You can add two variables together, divide them by 2, or multiply them by a third variable, and Python won't differentiate between variables and literals (like the number 2 in this example). However, performing arithmetic with variables doesn't change their values—you can only update a variable by using the equals sign (=).

Here's an example:

```
coffee_price = 2.50
number_of_coffees = 3
print(coffee_price * number_of_coffees)

Output: "7.5"

print(coffee_price)

Output: "2.5"

print(number_of_coffees)

Output: "3"

# Updating the price
coffee_price = 3.0

print(coffee_price * number_of_coffees)

Output: "9.0"

print(coffee_price)

Output: "3.0"

print(number_of_coffees)

Output: "3"
```

In this example, we first create two variables and assign numeric values to them. We then perform calculations using these variables. Notice that the variables themselves don't change as a result of these calculations. When we update the `coffee_price` variable and perform the calculations again, the program uses the updated value for `coffee_price` while the other variables remain the same.

2.1.9. Exponents

Python can also handle exponentiation, which is raising a number to a power. In math, you might see exponents as small numbers above a base number, but since typing those superscript numbers is tricky on most keyboards, Python uses instead. Here are some examples:

```
print(4 ** 8)
print(6 ** 4)
print(8 ** 3)
print(7 ** 0.5)

Output: 65536
Output: 1296
Output: 512
Output: 2.6457513110645907
```

2.1.10. Modulo

Python also has a tool called the modulo operator that works alongside division. The modulo operator, which is written as %, gives you the remainder when you divide two numbers. If the numbers divide evenly, the remainder will be 0.

Here are some more examples:

```
print(27 % 2)

Output: 1 because 27 divided by 2 is 13 with a remainder of 1

print(38 % 4)
Output: 2 because 38 divided by 4 is 9 with a remainder of 2
```

When you use modulo with 2, it returns 0 for even numbers and 1 for odd numbers

```
print(44 % 2) # Output: 0 because 44 is even
```

Here are some more examples:

```
print(5 % 2)  # Output: 1
print(6 % 2)  # Output: 0
print(7 % 2)  # Output: 1
print(8 % 2)  # Output: 0
print(9 % 2)  # Output: 1
```

Because of this repeating pattern, modulo is useful when you want to do something every nth time. For instance, if you own a restaurant and want to give a free meal to every 7th customer, you could use `customer_number % 7`—if the result is 0, that customer gets the free meal.

2.1.11. Concatenation

The `+` operator in Python doesn't just add numbers; it can also combine two strings! This process is called string concatenation. When you concatenate strings, you create a new string that's made up of the first string followed immediately by the second, with no space added between them.

Here's an example:

```
greeting_text = "Hello there,"
introduce_text = "my name is Johan."
full_text = greeting_text + introduce_text
print(full_text)

Output: "Hello there,my name is Johan"
```

In this code, we created two strings and then concatenated them. But as you can see, the result didn't include a space between the two sentences. To fix this, we can add a space using the same `+` operator

```
greeting_text = "Hello there,"
introduce_text = "my name is Johan."
full_text = greeting_text + " " + introduce_text
print(full_text)

Output: "Hello there, my name is Johan"
```

Or simply, add a space after the comma in `greeting_text`:

```
greeting_text = "Hello there,"
introduce_text = "my name is Johan."
full_text = greeting_text + " " introduce_text
print(full_text)

Output: "Hello there, my name is Johan"
```

Now, the output looks exactly as we expected.

If you want to concatenate a string with a number, you'll need to convert the number into a string first using the `str()` function. However, if you just want to print a numeric variable, you can use commas to pass it as a separate argument to the `print()` function without converting it.

Here's how that works:

```
birthday_string = "I am "
age = 18
birthday_string_2 = " years old today!"

# Convert the integer to a string before concatenating
full_birthday_string = birthday_string + str(age) + birthday_string_2
print(full_birthday_string)

Output: I am 18 years old today!

# Alternatively, print the number directly without converting
print(birthday_string, age, birthday_string_2)

Output: I am 18 years old today!
```

In the first example, we used `str()` to turn the integer age into a string so it could be concatenated with the other strings. In the second example, we didn't need to convert the number to a string because `print()` can handle different types of variables as separate arguments.

2.1.12. Plus Equals

Python provides a shorthand way to update variables with the += (plus-equals) operator. This is especially handy when you want to add something to the current value of a variable without rewriting the entire expression.

Here's how it works:

```
# Start with a variable holding a number
number_of_miles_ran = 8

# Now let's add to that number
# For example, we ran 2 more miles today
number_of_miles_ran += 2

# The new value is the old value plus the number after +=
print(number_of_miles_ran)

Output: 10
```

In this example, we keep a running total of the miles ran. Instead of recalculating the total from scratch each time, we simply update the existing total using the += operator.

You can also use += for string concatenation:

```
ordering_text = "Hi I'd like"
ordering_text += " a pizza"
ordering_text += " and a pepsi"
print(ordering_text)

Output: Hi I'd like a pizza and a pepsi
```

2.1.13. Multi-line Strings

Python strings are versatile, but if you try to create a string that spans multiple lines, you'll run into a `SyntaxError`. Luckily, Python has a solution: multi-line strings. By using triple quotes `"""` or `'''`, you can create a string that stretches across multiple lines. This method is particularly helpful when your string includes a lot of quotation marks, so you don't accidentally close the string too early.

Here's an example:

```
dreams_poem = """ Hold fast to dreams
For if dreams die
Life is a broken-winged bird
That cannot fly. """
```

2.1.14. Comments

When you write something in a program that you don't want the computer to run, it's called a comment. In Python, anything after a # is treated as a comment.

Here's how comments can be useful:

- Provide context: They explain why certain code is written a specific way.

```
# This variable will track how many times the word 'unicorn' appears in the text
unicorn_count = 0
```

- Help others understand your code: Comments can make it easier for others (or even yourself) to understand what the code does.

```
# This function predicts how many cookies we need to bake
cookie_baking_prediction()
```

- Temporarily ignore code: You can comment out a line to see how the program runs without it.

```
# backup_data = old_backup_method()
backup_data = new_backup_method()
```

2.1.15. User Input

So far, we've learned how to directly assign values to variables in a Python file. But what if we want the user to provide new information while the program is running?

This is where user input comes in. Instead of setting a value yourself, you can let the user enter the value using the `input()` function.

Just as we use `print()` to display a message, we use `input()` to get information from the user. The `input()` function takes a prompt message, which it shows to the user before they enter their response.

Here's an example:

```
favorite_color = input("What is your favorite color?")
print(favorite_color)

>>> What is your favorite color? Blue
>>> Blue
```

2.2. Functions

2.2.1. Introduction to Functions

When we start building bigger and more complicated programs, we'll often notice something: we need to repeat the same steps in different parts of our code. Instead of writing the same code over and over again, Python gives us a powerful tool to make our lives easier: functions.

Example: Food Delivery Application

Imagine we're creating a program for a food delivery app. Every time someone places an order, the app needs to:
- Confirm the order details.
- Calculate the delivery time based on the customer's location.
- Notify the customer about their delivery status.

These steps happen every time someone places an order. Instead of writing the same steps over and over, we can use a function to group these tasks into one reusable block of code.

What Are Functions?

A function is like a recipe in programming. It tells the computer what steps to follow. You can reuse this recipe as many times as you want without rewriting it!

Functions make your code:
- Shorter
- Easier to understand
- Simpler to update when needed

2.2.2. Defining a Function

A function has several parts, and the first step in learning about functions is understanding how to define one.

Here's what a basic function definition looks like in Python:

```
def function_name():
    # function tasks go here
```

Key Components of a Function

1. `def` **keyword:** This starts the function definition (it's called the function header).

2. **Function name:** A name in `snake_case` format that describes what the function does. It's a best practice to use descriptive yet short names.
3. **Parentheses** `()`: These can hold input values, called parameters. In this example, the function has no parameters.
4. **Colon** `:`: Marks the end of the function header.
5. **Function body:** This is where you write the steps the function will perform. All lines in the function body must be indented to show they belong to the function.

Here's an example of a function that welcomes a user to our food delivery app:

```
def delivery_welcome():
    print("Welcome to QuickEats!")
    print("We're excited to deliver your food.")
```
This function has:

A descriptive name, `delivery_welcome`, in `snake_case`. A body with two lines of code that print a welcome message.

Note: If you copy and paste the above function into your Python editor and click Run, you won't see anything happen yet. That's because defining a function doesn't automatically make it run. We'll learn how to use functions in the next exercise. For now, focus on writing and practicing function definitions!

2.2.3. Calling a Function

Now that we know how to define a function, let's learn how to call a function to make it run the code inside its body.

Calling a function means telling Python to execute the steps you've written in the function. To call a function, simply type its name followed by parentheses `()`.

Here's an example function that gives directions to pick up an order:

```
def pickup_directions():
    print("Walk 5 minutes to the nearest QuickEats hub.")
    print("Show your order confirmation at the counter.")
    print("Collect your order and enjoy!")
```

After defining the function, you can call it like this:

```
pickup_directions()
```

When you call the function, Python will execute the print statements inside the function's body, from top to bottom. The output will be:

```
Walk 5 minutes to the nearest QuickEats hub.
Show your order confirmation at the counter.
Collect your order and enjoy!
```

2.2.4. Whitespace & Execution Flow

Let's revisit our welcome function, this time for a food delivery application:

```
def delivery_welcome():
    print("Welcome to QuickEats!")
    print("We're excited to deliver your food.")
```

When the function `delivery_welcome()` is called, both print statements run together because they share the same indentation level (4 spaces).

In Python, whitespace (spaces or indentation) tells the computer which lines of code belong together. This is especially important for functions.

- Indented code: Part of the function body.
- Unindented code: Outside of the function body.

If we write another statement outside of the `delivery_welcome()` function, it must be unindented:

```
def delivery_welcome():
    # Indented code is part of the function body
    print("Welcome to QuickEats!")
    print("We're excited to deliver your food.")

# Unindented code below is outside the function body
print("Don't forget to check the daily discounts!")

delivery_welcome()
```

When you run the code above, the output will be:

```
Don't forget to check the daily discounts!
Welcome to QuickEats!
We're excited to deliver your food.
```

Why Does This Happen?

1. Python starts execution at the first line of code.
2. It executes line by line, from top to bottom. This is called execution flow.
3. The first print statement outside the function is executed immediately.
4. The function `delivery_welcome()` is called after that, so its print statements are executed next.

2.2.5. Parameters & Arguments

Let's revisit our food delivery example to make it more dynamic. Here's a simple welcome function:

```
def delivery_welcome():
    print("Welcome to QuickEats!")
    print("Looks like your order is coming from Joe's Pizza.")
```

If we call this function: `delivery_welcome()`

The output is:

```
Welcome to QuickEats!
Looks like your order is coming from Joe's Pizza.
```

But what if the order isn't from Joe's Pizza? This function works only for that specific case. To make it flexible, we can use function parameters.

What Are Parameters?

Parameters allow a function to take input and use it in its body. They're placeholders for the data the function will receive. Let's improve our function with a parameter:

```
def delivery_welcome(restaurant):
    print("Welcome to QuickEats!")
    print("Looks like your order is coming from " + restaurant + ".")
```

What About Arguments?

When we call the function, we provide arguments (the actual data) to fill in the parameter. Here's how:

```
delivery_welcome("Joe's Pizza")
```

This outputs:

```
Welcome to QuickEats!
Looks like your order is coming from Joe's Pizza.
```

If we want to change the restaurant, we just pass a different argument: `delivery_welcome("Sushi World")`

The new output is:

```
Welcome to QuickEats!
Looks like your order is coming from Sushi World.
```

Key Points

1. Parameters: Names you define in the function to accept input.

Example: restaurant in delivery_welcome(restaurant).

2. Arguments: The actual data you pass to the function when calling it.

Example: "Joe's Pizza" or "Sushi World".

2.2.6. Multiple Parameters

So far, we've seen how to use one parameter in a function, but functions can take multiple parameters to handle more inputs.

You can add more than one parameter to your function by separating them with commas:

```
def my_function(parameter1, parameter2, parameter3):
    # Some code
```

When calling the function, you'll need to pass arguments for each parameter:

```
my_function(argument1, argument2, argument3)
```

Let's enhance our food delivery welcome function to include both the customer's name and the restaurant they ordered from:

```
def delivery_welcome(customer_name, restaurant):
    print("Welcome to QuickEats, " + customer_name + "!")
    print("Your order from " + restaurant + " is on its way.")
```

In this function:

`customer_name` and `restaurant` are the two parameters. They allow the function to include both the customer's name and the restaurant in the message.

When calling the function, make sure to pass arguments for both parameters in the same order as defined: `delivery_welcome("Alice", "Joe's Pizza")`

This will output:

```
Welcome to QuickEats, Alice!
Your order from Joe's Pizza is on its way.
```

If you call the function with different arguments, like this:

```
delivery_welcome("Bob", "Sushi World")
```

The output will change accordingly:

```
Welcome to QuickEats, Bob!
Your order from Sushi World is on its way.
```

Key Points

1. Order matters: The arguments you pass will match the parameters in the order they are listed.
2. Use commas: Separate multiple parameters with commas in both the function definition and the function call.
3. Flexibility: Adding parameters makes your function more versatile.

2.2.7. Types of Arguments

In Python, functions can take three main types of arguments:

1. Positional Arguments: Assigned based on their position in the function call.
2. Keyword Arguments: Assigned explicitly by their name in the function call.
3. Default Arguments: Given a default value in the function definition.

Positional Arguments

These are the most common type of arguments and are assigned based on the order in the function definition.

Example:

```
def calculate_delivery_cost(distance, rate, discount):
    print(distance * rate - discount)
```

When calling this function, the order of arguments matters:

```
calculate_delivery_cost(5, 2, 1)
# 5 is distance, 2 is rate, 1 is discount

Output: 9
```

Keyword Arguments

With keyword arguments, you explicitly state which argument matches which parameter. The order doesn't matter.

Example:

```
calculate_delivery_cost(rate=2, discount=1, distance=5)
```

This will still output: 9

Default Arguments

Default arguments allow you to set a default value for a parameter in the function definition. If no value is provided during the function call, the default is used.

Example:

```
def calculate_delivery_cost(distance, rate, discount=3):
    print(distance * rate - discount)
```

Now, you can call the function in two ways:

Without specifying a discount (uses the default value of 3):

```
calculate_delivery_cost(5, 2)

Output: 7
```

By overriding the default value with your own argument:

```
calculate_delivery_cost(5, 2, 1)
Output: 9
```

Key Points

1. Positional Arguments: Order matters when calling the function.
2. Keyword Arguments: Specify the parameter name for flexibility in order.
3. Default Arguments: Provide default values for parameters when defining the function.

2.2.8. Built-in Functions vs. User-Defined Functions

In Python, functions fall into two main categories:

1. User-Defined Functions: Functions we write ourselves.
2. Built-in Functions: Functions that Python provides for us to use.

User-Defined Functions

These are the functions we've been creating in our lessons. For example:

```
def greet_user(name):
    print("Hello, " + name + "!")
```

These functions are written by us to perform specific tasks.

Built-in Functions

Python also comes with many built-in functions that are ready to use. You've already been using some, like `print()` and `str()`.

Here's another example with the built-in function `len()`:

```
destination_name = "Mount Everest"

# Built-in function: len()
length_of_name = len(destination_name)

# Built-in function: print()
print(length_of_name)

Output:13
```

In this example:

`len()` : Calculates the length of the string “Mount Everest”.

`print()` : Outputs the result.

2.2.9. Variable Access

As our programs grow with more functions, it’s important to understand where variables can be accessed. This is determined by a variable’s scope.

Let’s revisit this example:

```
def delivery_message(restaurant):
    print("Your order from " + restaurant + " is on its way!")
```

What happens if we try to access the variable restaurant outside of the function?

```
def delivery_message(restaurant):
    print("Your order from " + restaurant + " is on its way!")

print(restaurant)
```

When you run this code, you’ll get an error. That’s because the scope of `restaurant` is limited to the function `delivery_message()`. Variables defined inside a function can only be accessed within that function.

Variables Outside a Function

Now let’s define a variable outside of any function and see what happens:

```
delivery_fee = 5

def delivery_message(restaurant="Sushi World"):
    print("Your order from " + restaurant + " is on its way!")
    print("Your delivery fee is $" + str(delivery_fee))

print(delivery_fee)
delivery_message()

Output: 5
```

Your order from Sushi World is on its way! Your delivery fee is $5

Here’s what’s happening:

The variable `delivery_fee` is defined outside of the function. This makes it accessible both inside and outside the function.

The function `delivery_message()` successfully uses `delivery_fee` because it's defined globally.

Key Takeaways

1. Variables defined inside a function can only be accessed within that function.

- These variables are called local variables.

1. Variables defined outside any function can be accessed from anywhere in the file.

- These variables are called global variables.

Practice

Play around with defining variables inside and outside of functions. Experiment to see where they can and cannot be accessed! Understanding scope is essential for writing clear and organized programs.

2.2.10. Returns

So far, our functions have been using `print()` to show output. But functions can also return values to the program using the Python keyword return. This allows the value to be used or modified later in the program.

Here's an example of a function that calculates a tip for a food delivery order:

```
def calculate_tip(total_amount, tip_percentage):
    return total_amount * (tip_percentage / 100)
```

We can store the returned value in a variable to use it later:

```
order_tip = calculate_tip(50, 15)

print("A 15% tip on a $50 order is $" + str(order_tip))
Output: A 15% tip on a $50 order is $7.5
```

Why Use return?

1. When a function returns a value, that value can be:
2. Stored in a variable (like `order_tip`).
3. Used in calculations or as input for other functions.

Displayed later in the program when needed.

For example:

```
final_total = 50 + order_tip
print("Your final total is $" + str(final_total))

Output: Your final total is $57.5
```

Key Points

1. Return Values:

- Functions use return to send a value back to the program.
- Returned values can be stored in variables for reuse.

2. Returned Function Value:

- When a function's result is saved in a variable, it's called a returned function value.

3. Reuse and Modify:

- Returned values make your programs more flexible and efficient.

2.3. Control Flow

2.3.1. Introduction to Control Flow

Imagine playing a video game.

You start a new game and think: "Do I have enough coins to unlock the next level?"

If you have enough coins, you unlock the level and keep playing. If not, you decide to play a mini-game to earn more coins. After unlocking the level, you check: "Is my health full? Do I need to use a health potion?"

Each decision affects your gameplay. Similarly, a computer uses control flow to determine which actions to take based on conditions.

Control flow refers to the order in which your program runs. In Python:
1. The program starts at the top of the code.
2. It executes line by line.
3. Conditional statements (like if, else) act as "checkpoints" to decide which path to take.

Here's how your gaming scenario might look in Python:

```
coins = 10
health = 50
health_full = 100

if coins >= 20:
    print("Unlock the next level!")
else:
    print("Play a mini-game to earn more coins.")

if health < health_full:
    print("Use a health potion to restore health.")
else:
    print("Your health is already full. No action needed.")
```

Why is Control Flow Important?

Control flow allows programs to:
1. Make decisions: Just like deciding whether to unlock a level or earn more coins.
2. Take different paths: Depending on the conditions set.
3. Execute logically: Ensuring the program follows the right steps.

In the upcoming lessons, you'll learn to use conditional statements like `if`, `else`, and `elif` to control the flow of your programs, making them interactive and intelligent!

2.3.2. Boolean Expressions

To build control flow in programs, we often need to check if a condition is true or false. For this, we use Boolean expressions.

A Boolean expression is a statement that can only be answered with **true** or **false**.

It must:
- Be verifiable with evidence.
- Avoid relying on opinions or interpretations.

Imagine checking if a classroom is available for a meeting:

```
classroom_available = True
```

This is a Boolean expression because it can only be either:
- True if the classroom is available.
- False if it is not.

It's verifiable and doesn't depend on anyone's opinion.

Consider the statement: "Math is the best subject."

This is not a Boolean expression because it's an opinion. Someone else might think science is better, and both viewpoints would be subjective.

Let's look at this question:

"Are there more than 20 students in the class?"

This is a Boolean expression because it can be verified. For instance:

```
students_in_class = 25
is_crowded = students_in_class > 20
```

Here, `is_crowded` evaluates to **True** because 25 > 20.

Even if the result is **False**, it's still a Boolean expression because it meets the criteria: it's verifiable and has only two possible outcomes.

2.3.3. Relational Operators

Now that we understand Boolean expressions, let's learn how to create them using relational operators. These operators compare two values and return True or False based on the comparison.

Relational operators (also called comparison operators) allow us to compare values. They evaluate to True or False depending on the relationship between the operands.

Equals to `==` and Not Equals to `!=`

The first two relational operators we'll explore are:
1. Equals to `==`:

- Returns True if the two values are the same.
- Returns False if the two values are different.

2. Not equals to `!=`:

- Returns True if the two values are different.
- Returns False if the two values are the same.

Here's how these operators work in Python:

```
1 == 1       # True: Both values are the same
1 != 1       # False: Both values are the same
2 != 4       # True: The values are different
3 == 5       # False: The values are different
'7' == 7     # False: Different types (string vs number)
```

2.3.4. If Statement

Understanding Boolean variables and expressions is essential because they are the foundation of conditional statements.

Conditional statements let your program make decisions based on conditions.

Imagine checking your exam results: If your score is above 70, then you pass.

Here, the Boolean expression is:

"Your score is above 70."

If this expression evaluates to True, you pass the exam. If it's False, you don't.

Here's how this example looks in Python:

```
score = 85

if score > 70:
    print("You passed!")
```

The if keyword starts the conditional statement. A colon `:` signals the start of the block of code that runs if the condition is True.

The block of code is indented under the if statement . In this case, since the score is 85 (which is greater than 70), the program will output: `You passed!`

Here's a simple numeric check:

```
if 10 > 5:
    print("10 is greater than 5")
```

Let's break it down:
- The condition 10 > 5 evaluates to True.
- Since the condition is met, the block of code runs, and the output is:

```
10 is greater than 5
```

2.3.5. Relational Operators II

Now that we've learned how to use conditional statements, let's explore additional relational operators for building Boolean expressions. Previously, we covered `==` (equals) and `!=` (not equals). Now, let's add these four operators to our toolkit:

- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

Imagine we want to check if a classroom has enough seats for students:

```
students = 25
seats = 30

if students <= seats:
    print("There are enough seats for all students.")
```

Here's what happens:

The condition students `<=` seats evaluates to **True** because 25 is less than or equal to 30. The program prints:

```
There are enough seats for all students.
```

Let's say you're creating a game and want to ensure only players aged 18 or older can access it:

```
player_age = 16

if player_age < 18:
    print("Access denied. You must be 18 or older to play.")

Output:
Access denied. You must be 18 or older to play.
```

Here, the condition `player_age < 18` evaluates to **True** because 16 is less than 18.

2.3.6. Boolean Operators: and

Sometimes, you'll want to check multiple conditions in your if statements. In these cases, you can combine smaller Boolean expressions using Boolean operators (also known as logical operators).

Python has three Boolean operators:
- and: True if both conditions are True.
- or: True if at least one condition is True.
- not: Negates a condition (True becomes False and vice versa).

Let's focus on the `and` operator.

The `and` operator evaluates to **True** only if both of the conditions it combines are **True**. If either condition is **False**, the entire expression is **False**.

Imagine you're checking if a library visitor meets two rules:
- They have a library card.
- They returned their overdue books.

```
has_library_card = True
returned_books = False

if has_library_card and returned_books:
    print("You can borrow new books.")
else:
    print("Please make sure to meet all requirements.")
```

Explanation:

`has_library_card` is **True**, but `returned_books` is **False**. Since `and` requires both conditions to be **True**, the entire expression is **False**, and the program outputs:

```
Please make sure to meet all requirements.
```

More examples:

```
(3 > 1) and (5 == 5)   # True: Both parts are True
(2 < 1) and (5 != 6)   # False: First part is False
(1 == 1) and (2 > 3)   # False: Second part is False
(0 > 10) and (1 == 0)  # False: Both parts are False
```

2.3.7. Boolean Operators: or

The or operator combines two Boolean expressions into a single, larger expression. It evaluates to **True** if at least one of the components is **True**. If both components are **False**, the whole expression is **False**.

Consider the statement:

“Oranges are a fruit or apples are a vegetable.”

This expression consists of two smaller Boolean expressions:
1. “Oranges are a fruit” – This is True.
2. “Apples are a vegetable” – This is False.

Since the or operator is used, the overall statement is True, because at least one of the components is True.

Important Note: In Python, or does not require one component to be False for the other to be True. If both components are True, the expression is still True.

Here are some examples of how or works:

```
True or (3 + 4 == 7)    # True: First part is True
(1 - 1 == 0) or False   # True: First part is True
(2 < 0) or True         # True: Second part is True
(3 == 8) or (3 > 4)     # False: Both parts are False
```

2.3.8. Boolean Operators: not

The not operator is the simplest Boolean operator. It reverses the truth value of a Boolean expression:
- If the expression is True, applying `not` makes it False.
- If the expression is False, applying `not` makes it True.

Consider the statement: “Oranges are not a fruit.”

Here, we started with the True statement: “Oranges are a fruit”

Then, by applying not, we reversed its truth value to: “Oranges are not a fruit” (False).

In Python, the not operator is placed at the beginning of the Boolean expression. Here’s how `not` works in Python:

```
not True            # False
not False           # True
not 1 + 1 == 2      # False: 1 + 1 == 2 is True, and not reverses it
not 7 < 0           # True: 7 < 0 is False, and not reverses it
```

2.3.9. Else Statements

When you start adding many `if` statements, your code can become cluttered. To simplify it, Python provides the `else` statement.

The `else` statement lets you specify what your program should do when the condition in the `if` statement is not met.

An `else` statement must always follow an `if` statement. It acts as a “default” block of code, executed when the `if` condition is **False**.

Example: Morning Routine

```
if weekday:
    print("Wake up at 6:30.")
else:
    print("Sleep in.")
```

If weekday is True, the program prints: `Wake up at 6:30.`

If weekday is False, the program prints: `Sleep in.`

This eliminates the need for multiple `if` statements for every possible condition. Let’s say you’re creating a program to check if a student passed or failed:

```
score = 45

if score >= 50:
    print("You passed!")
else:
    print("You failed. Better luck next time.")
```

If the score is 50 or higher, the program outputs: `You passed!`

Otherwise, it outputs: `You failed. Better luck next time.`

2.3.10. Else If Statements (elif)

In addition to `if` and `else`, Python provides `elif` statements, short for *else if*. An `elif` statement allows your program to check additional conditions after the first `if` condition is not met.

The program checks conditions in this order:
1. Start with the if statement.
2. If the if condition is False, check the conditions in each elif statement from top to bottom.

3. If none of the if or elif conditions are met, execute the else block (if it exists).

This structure ensures that only one block of code is executed.

Let's write a program that assigns grades based on a student's score:

```
score = 78

if score >= 90:
    print("You got an A!")
elif score >= 80:
    print("You got a B!")
elif score >= 70:
    print("You got a C!")
elif score >= 60:
    print("You got a D!")
else:
    print("You got an F.")
```

Here's what happens:

The program checks if the score is 90 or above (`if score >= 90`). Since 78 is less than 90, this condition is **False** .

It moves to the first `elif` (`elif score >= 80`) and checks if 78 is greater than or equal to 80. This is also **False**.

It moves to the next `elif` (`elif score >= 70`). Since 78 is greater than 70, this condition is True, and it prints:

```
You got a C!
```

No further conditions are checked after this point.

Why Use elif?

If you replaced the `elif` statements with `if` statements, the program would check every condition, even `if` one is already met.

For example:

```
score = 95

if score >= 90:
    print("You got an A!")
if score >= 80:
    print("You got a B!")
```

This would print both messages because both conditions are **True**. By using `elif` , only one block of code executes, making your program more efficient and predictable.

2.4. Manipulating Data in Python

2.4.1. What is a List?

In programming, we often need to work with collections of data. A list in Python is a built-in data structure that allows us to store multiple items in a specific order.

Suppose you want to create a list of your favorite fruits:
- Mango
- Apple
- Banana
- Orange

We can store these values in a Python list like this:

```
favorite_fruits = ["Mango", "Apple", "Banana", "Orange"]
```

Key Features of Lists

1. Square Brackets:

- A list starts and ends with square brackets: `[ ]`.

2. Items:

- Each item in the list (like "Mango" or "Apple") is separated by a comma.

3. Quotation Marks:

- Strings in a list (like the fruit names) are enclosed in quotation marks: `" "`. For numbers, quotation marks are not needed.

4. Spacing:

- It's good practice to add a space after each comma for better readability:

```
favorite_fruits_and_one_random_num = ["Mango", "Apple", "Banana", 14]
```

Python will still run your code even without spaces, but it will be harder to read.

Why Use Lists?

Lists make it easy to store and manage multiple related items. For example, with the `favorite_fruits` list, you can:
1. Access specific items: Find your second favorite fruit.
2. Add or remove items: Update the list when your preferences change.
3. Perform operations: Count how many fruits you like or sort them alphabetically.

2.4.2. What Can a List Contain?

Lists in Python can store much more than just numbers. They are flexible and can hold a variety of data types.

Instead of storing numbers, you can create a list of strings. For example, a list of pet names:

```
pet_names = ["Buddy", "Luna", "Charlie", "Max"]
```

This list contains strings representing the names of pets.

Lists can hold multiple data types. For example, a list combining a string and a number:

```
mixed_list_string_number = ["Buddy", 5]
```

- "Buddy" is a string.

- 5 is an integer.

You can include a mix of data types like strings, numbers, booleans, and floats:

```
mixed_list_common = ["Luna", 4, True, 2.5]
```

This list contains:
- "Luna" (string)
- 4 (integer)
- True (boolean)
- 2.5 (float)

2.4.3. List Methods

In Python, lists come with built-in tools called methods. Methods are special functions designed to work with a specific data type (like lists) to help us create, modify, or delete data.

A method is like a built-in tool for lists. It's written in the format: `list_name.method()`

Some methods take an input value inside the parentheses, while others don't.

2.4.4. Increasing List Size: Append Method

The `.append()` method adds a new element to the end of an existing list. This is helpful when you want to grow your list dynamically.

For example:

```
example_list = ["apple", "banana", "cherry"]
example_list.append("date")
print(example_list)

Output: ['apple', 'banana', 'cherry', 'date']
```

Here's what happens:

`.append("date")` adds "date" to the end of `example_list`. The list is updated to include the new element.

2.4.5. Increasing List Size: Using the `+` Operator

When you need to add multiple items to a list, you can use the `+` operator to concatenate (combine) two lists. Suppose a bakery has a list of items it sells:

```
items_sold = ["cake", "cookie", "bread"]
```

The bakery decides to start selling "biscuit" and "tart". You can create a new list that combines both old and new items:

```
items_sold_new = items_sold + ["biscuit", "tart"]
print(items_sold_new)

Output: ['cake', 'cookie', 'bread', 'biscuit', 'tart']
```

The original list `items_sold` does not change:

```
print(items_sold) # Output: ['cake', 'cookie', 'bread']
```

This is because the `+` operator creates a new list without modifying the original.

If you want to add a single element using +, you must place it inside square brackets ([]) to turn it into a list. For example:

```
my_list = [1, 2, 3]
my_new_list = my_list + [4]
print(my_new_list)

Output: [1, 2, 3, 4]
```

If you try to add a single element without brackets, you'll get an error:

```
my_new_list = my_list + 4

TypeError: can only concatenate list (not "int") to list
```

2.4.6. Accessing List Elements

When working with lists in Python, you often need to access specific elements. Each element in a list is assigned a position, known as its index.

Python lists are zero-indexed, meaning:
- The first element is at index 0.
- The second element is at index 1.
- The third element is at index 2, and so on.

Let's say you're planning your meals for the week and have the following list:

```
meals = ["Pasta", "Salad", "Tacos", "Pizza", "Soup"]
```

Here's the index breakdown:

Element	Index
"Pasta"	0
"Salad"	1
"Tacos"	2
"Pizza"	3
"Soup"	4

Accessing Elements by Index

To retrieve a specific item from a list in Python, use square brackets `[]` with the index of the element you want to access.

Imagine you have a list of books you plan to read:

```
books = ["1984", "Pride and Prejudice", "The Great Gatsby", "To Kill a Mockingbird",
"Moby Dick"]
```

To access the fourth book:

```
print(books[3])
Output: To Kill a Mockingbird
```

Here, `books[3]` retrieves the element at index 3, which is "To Kill a Mockingbird". Remember, indexing starts at 0.

Important Notes on Indexing:
1. Index Must Be an Integer

- You cannot use non-integer indices. For example:

```
print(books[4/2])  # Causes an error because 4/2 evaluates to 2.0 (a float)

TypeError: list indices must be integers or slices, not float
```

You can fix this issue by converting the float to an integer with int():

```
print(books[int(4/2)]) # Output: The Great Gatsby
```

2.4.7. Accessing List Elements: Negative Index

When you need to select the last element of a list, Python provides an easy way to do so using negative indexing. Negative indices count from the end of the list, with `-1` representing the last element. Imagine you have a list of cities:

```
cities = ["New York", "Tokyo", "Paris", "Berlin", "Sydney", "Rio"]
```

To access the last city in the list:

```
print(cities[-1]) # Output: Rio
```

Here, `cities[-1]` retrieves `Rio`, the last element in the list.

Using negative indices is equivalent to using positive indices for the same position. In the above list:

- `cities[-1]` gives the last element `Rio.`
- `cities[5]` also gives the last element `Rio` because the list has 6 items, and the last index is 5.

```
print(cities[5]) # Output: Rio
```

Here's how negative indexing exactly work for the cities list:

Element	Positive Index	Negative Index
"New york"	0	−6
"Tokyo"	1	−5
"Paris"	2	−4
"Berlin"	3	−3
"Sydney"	4	−2
"Rio"	5	−1

Why Use Negative Indices?

Negative indices are especially useful when:
- You don't know the exact length of the list.
- You want to access elements starting from the end of the list.

2.4.8. Modifying List Elements

Lists in Python are mutable, which means you can change their elements after they are created. To modify an item in a list, use its index to reassign its value.

Suppose you have a list of favorite movies:

```
movies = ["Inception", "Interstellar", "Titanic", "Avatar"]
```

You realize that "Titanic" is no longer a favorite, and you want to replace it with "The Matrix". Here's how:

```
movies[2] = "The Matrix"
print(movies)

Output: ["Inception", "Interstellar", "The Matrix", "Avatar"]
```

Here's what happened:
- The element at index 2 ("Titanic") was replaced with "The Matrix".
- The list is now updated.

You can also use negative indices to modify list elements. For example, to replace the last movie "Avatar" with "The Godfather":

```
movies[-1] = "The Godfather"
print(movies)

Output: ["Inception", "Interstellar", "The Matrix", "The Godfather"]
```

Here, `movies[-1]` refers to the last element, which was updated.

2.4.9. Two-Dimensional (2D) Lists

In Python, lists can contain other lists, forming what is commonly referred to as a two-dimensional (2D) list. This structure is especially useful for organizing data in a grid-like format, such as tables or game boards.

Imagine we're tracking students and their grades:
- Alex scored 85.
- Jamie scored 90.
- Taylor scored 78

. We can represent this data as a 2D list, where each inner list contains a student's name and their grade:

```
grades = [["Alex", 85], ["Jamie", 90], ["Taylor", 78]]
```

To access elements in a 2D list, use two indices:
- The first index selects the inner list.
- The second index selects the element within that inner list.

Element	Positive Index
“Alex”	grades[0][0]
85	grades[0][1]
“Jamie”	grades[1][0]
90	grades[1][1]
“Taylor”	grades[2][0]
78	grades[2][1]

Example: Get Jamie’s Grade

```
print(grades[1][1]) # Output:90
```

Here’s what happens:

- `grades[1]` selects the second list: `["Jamie", 90]`.
- `[1]` selects the second element within that list: `90`.

A 2D list is perfect for representing a grid, such as a Tic-Tac-Toe board:

```
tic_tac_toe = [
    ["X", "O", "X"],
    ["O", "X", "O"],
    ["O", "O", "X"]
]
```

To get the middle element (“X” in the second row): `print(tic_tac_toe[1][1]) # Output: X`

2.4.10. Modifying 2D Lists

Now that we know how to access elements in a 2D list, modifying them is just as simple. You use the specific index to reassign a new value.

Suppose we have a list where each student’s name and grade are stored:

```
grades = [["Alice", 85], ["Bob", 92], ["Charlie", 78]]
```

If “Alice” improves her grade to 90, you can update her grade like this:

```
grades[0][1] = 90
print(grades)

Output: [["Alice", 90], ["Bob", 92], ["Charlie", 78]]
```

Explanation:

- `grades[0]` accesses the first inner list: `["Alice", 85]`.
- `[1]` accesses the second item in that list: `85`.
- The value is updated to `90`.

You can also use negative indices to modify elements. For example, to change “Charlie”’s grade to 80:

```
grades[-1][-1] = 80
print(grades)
Output: [["Alice", 90], ["Bob", 92], ["Charlie", 80]]
```

Explanation:
- `grades[-1]` accesses the last list: `["Charlie", 78]`.
- `[-1]` accesses the last element in that list: `78`.
- The value is updated to `80`.

2.5. Manipulate Lists

2.5.1. Using Lists

Now that we know how to create and access list data, we can explore additional ways to work with lists in Python.

In this lesson, you'll discover how to:
1. Add and Remove Items: Insert or delete items at specific positions.
2. Create Continuous Lists: Generate lists with sequential values.
3. Find the Length: Use the len() function to determine how many items are in a list.
4. Slice Lists: Select specific portions of a list.
5. Count Elements: Use the .count() method to see how often an item appears.
6. Sort Lists: Organize items in ascending or descending order.

Methods vs. Built-In Functions

Python provides two ways to work with lists:

Methods: Called directly on a list using the syntax: `list.method(input)`

Example: Adding an item using .append():

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

Built-In Functions: Standalone functions applied to lists using the syntax:

`builtinfunction(input)`

Example: Finding the length of a list with len():

```
my_list = [1, 2, 3]
print(len(my_list))  # Output: 3
```

2.5.2. Adding by Index: Using .insert()

The `.insert()` method in Python allows you to add an element to a specific index in a list. This is useful when you need to place an item in a specific position rather than just appending it to the end.

The .insert() method takes two inputs:
- Index: The position where you want to insert the element.
- Element: The value you want to insert.

Imagine you're organizing a playlist of songs:

`playlist = ["Song A", "Song C", "Song D"]`

You realize you forgot "Song B" and want to insert it between "Song A" and "Song C":

```
playlist.insert(1, "Song B")
print(playlist)

Output: ["Song A", "Song B", "Song C", "Song D"]
```

Here's what happened:
1. `.insert(1, "Song B")` specifies index 1 (between "Song A" and "Song C") for the new element.
2. "Song B" is added, and the other elements shift to the right.

You can also use negative indices with `.insert()`. For example, if you want to insert "Song E" just before the last song:

```
playlist.insert(-1, "Song E")
print(playlist)

Output: ["Song A", "Song B", "Song C", "Song E", "Song D"]
```

2.5.3. Removing by Index: Using .pop()

The `.pop()` method in Python allows you to remove an element from a list at a specific index. It is useful when you want to delete an element and optionally keep track of the value that was removed.

The `.pop()` method takes one optional input:

Index: The position of the element to remove. If no index is specified, `.pop()` removes the last element by default.

Suppose you have a to-do list:

`tasks = ["Email client", "Submit report", "Water plants", "Pick up groceries"]`

If you finished the last task, "Pick up groceries", you can remove it like this:

```
removed_task = tasks.pop()
print(tasks)
print(removed_task)

Output:

["Email client", "Submit report", "Water plants"]
"Pick up groceries"
```

Explanation:
1. `.pop()` removes the last element.
2. It returns the removed value, which we store in `removed_task`.

Suppose you decide not to "Water plants" (at index 2):

```
tasks.pop(2)
print(tasks)

Output: ["Email client", "Submit report", "Pick up groceries"]
```

Explanation:

- `.pop(2)` removes the element at index 2.
- The list is updated, and no value is stored since we didn't assign the `.pop()` result to a variable.

What If the Index Doesn't Exist?

If you try to remove an element at an invalid index or use `.pop()` on an empty list, Python raises an `IndexError`:

```
tasks.pop(5)  # Invalid index

IndexError: pop index out of range
```

2.5.4. Consecutive Lists: Using range()

Creating a list of consecutive numbers is a common task in programming. Instead of typing all the numbers manually, Python provides the `range()` function to generate these numbers easily.

The `range()` function takes a single input and generates numbers starting at 0 and ending at the number before the input.

Example: Numbers from 0 to 9

Instead of writing:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can use:

```
my_range = range(10)
print(my_range)

Output: range(0, 10)
```

Converting a Range to a List

The `range()` function creates a range object, not a list. To use it as a list, convert it using the `list()` function:

```
print(list(my_range)) Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Suppose you're assigning seat numbers for a classroom with 15 seats:

```
seat_numbers = list(range(1, 16))
print(seat_numbers)

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

2.5.5. Unlocking the Full Potential of range()

By default, the `range()` function generates numbers starting at 0. However, with additional inputs, you can customize the start point, endpoint, and even the step size of the sequence.

If you provide two inputs to `range()`, you can set the starting point and the endpoint (exclusive).

Example: Start at 2, End at 8

```
my_list = range(2, 9)
print(list(my_list))

Output: [2, 3, 4, 5, 6, 7, 8]
```

Here, the sequence starts at 2 and ends just before 9.

Skipping Numbers with Steps

Adding a third input allows you to set the step size, which determines how much to "skip" between numbers.

Example: Skip by 2

```
my_range2 = range(2, 9, 2)
print(list(my_range2))

Output: [2, 4, 6, 8]
```

The sequence starts at 2 and increments by 2 until it reaches (but doesn't exceed) 9.

Example: Larger Steps

You can skip more numbers by increasing the step size.

```
my_range3 = range(1, 100, 10)
print(list(my_range3))

Output: [1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
```

Here's what happens:
- The sequence starts at 1.
- Each number is 10 greater than the previous.
- It stops before reaching 100.

2.5.6. Finding the Length of a List

In Python, knowing how many items are in a list is often essential. The `len()` function provides a simple way to calculate the number of elements in a list.

The `len()` function takes a list (or another iterable) as input and returns the total number of items in it. Suppose you have a list of your top five hobbies:

```
hobbies = ["Reading", "Gaming", "Hiking", "Cooking", "Traveling"]
```

To find out how many hobbies are in the list:

```
print(len(hobbies)) # Output: 5
```

2.5.7. Slicing Lists: Extracting Portions of a List

In Python, slicing allows you to extract specific portions of a list. This is useful when you only need part of the data without modifying the original list. The syntax for slicing is:

`list[start:end]`

- start: The index of the first element you want to include.
- end: The index just beyond the last element you want to include (non-inclusive).

Suppose you have a list of letters:

```
letters = ["a", "b", "c", "d", "e", "f", "g"]
```

If you want to extract "b" through "f", you would use:

```
sliced_list = letters[1:6]
print(sliced_list)

Output: ["b", "c", "d", "e", "f"]
```

Explanation:
- The slicing starts at index 1 ("b").
- It ends at index 6, but the element at 6 ("g") is not included.

Leaving Out `start` or `end`

You can omit either start or end to slice from the beginning or to the end of the list:

From the beginning:

```
print(letters[:4])  # ["a", "b", "c", "d"]
```

To the end:

```
print(letters[3:])  # ["d", "e", "f", "g"]
```

2.5.8. Counting Items in a List

In Python, you can easily count how many times an item appears in a list using the `.count()` method. This is particularly helpful for analyzing data or identifying trends in a list.

The `.count()` method takes a single input (the item you want to count) and returns the number of times it appears in the list. Suppose we have a list representing the letters in the word "Mississippi":

```
letters = ["m", "i", "s", "s", "i", "s", "s", "i", "p", "p", "i"]
```

To count how many times "i" appears:

```
num_i = letters.count("i")
print(num_i)

Output: 4
```

Explanation:
- `.count("i")` checks the list for every occurrence of "i" and returns the total.

Counting in a 2D List

You can also use `.count()` to count the occurrences of sublists in a two-dimensional list. For example:

```
number_collection = [[100, 200], [100, 200], [475, 29], [34, 34]]
```

To find out how many times `[100, 200]` appears:

```
num_pairs = number_collection.count([100, 200])
print(num_pairs)

Output: 2
```

2.5.9. Sorting Lists in Python

Sorting is a common operation when working with lists. Python's `.sort()` method allows you to arrange elements in a list in either ascending or descending order.

The `.sort()` method modifies the original list and does not return any value. It works for both numerical and alphabetical lists. Suppose you have a list of names:

```
names = ["Xander", "Buffy", "Angel", "Willow", "Giles"]
```

To sort these names in alphabetical order:

```
names.sort()
print(names)

Output: ['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

Explanation:

`.sort()` arranges the list in ascending (A-Z) order for strings. You can sort the list in reverse order by setting the reverse parameter to **True**:

```
names.sort(reverse=True)
print(names)

Output: ['Xander', 'Willow', 'Giles', 'Buffy', 'Angel']
```

Sorting works just as well for numerical lists:

```
numbers = [42, 16, 8, 23, 15]

# Ascending order
numbers.sort()
print(numbers)

Output: [8, 15, 16, 23, 42]

# Descending order
numbers.sort(reverse=True)
print(numbers)

Output: [42, 23, 16, 15, 8]
```

2.5.10. Exploring the Python zip()

Python's `zip()` function is a powerful tool for combining related data from multiple lists. It allows you to efficiently create pairs or groupings of elements from separate lists without relying on multi-dimensional structures.

The `zip()` function takes two or more lists as inputs and returns a zip object containing pairs of corresponding elements from the lists.

Suppose you have a list of student names and their respective heights:

```
names = ["Jenny", "Alexus", "Sam", "Grace"]
heights = [61, 70, 67, 64]
```

To combine these into a list of name-height pairs:

```
names_and_heights = zip(names, heights)
```

At this point, `names_and_heights` is a zip object that represents the pairs but isn't directly readable. To make the zip object readable, convert it into a list using the `list()` function:

```
converted_list = list(names_and_heights)
print(converted_list)

Output: [('Jenny', 61), ('Alexus', 70), ('Sam', 67), ('Grace', 64)]
```

1. The output is a list of tuples, not lists. Tuples are immutable (unchangeable), which makes them useful for fixed data sets.
2. Each tuple contains one element from each input list in the same order.

2.6. Learn about Loops

2.6.1. Introduction to Loops

In our daily lives, we often repeat tasks until a specific goal is achieved. This repetition is mirrored in programming using loops, which allow us to execute a set of actions multiple times.

Imagine packing a suitcase:
1. Initialization: You have a pile of clothes to pack and start with the first item.
2. Repetition: You take one piece of clothing, fold it, and place it in the suitcase. Repeat for the next item.
3. End Condition: When the pile is empty, you stop packing.

What Is a Loop in Programming?

A loop is a way to repeat a task in programming by following three components:
1. Initialization: Start the process.
2. Repetition: Perform the task repeatedly.
3. End Condition: Stop when a specific condition is met.

This process of repeating tasks is called iteration.

Python supports two main types of loops:
1. Indefinite Iteration: The number of repetitions depends on a condition.

- Example: Keep looping until a user guesses the correct number.

2. Definite Iteration: The number of repetitions is predefined.

- Example: Loop through every item in a list.

2.6.2. For Loops: An Introduction

A for loop is a type of definite iteration, meaning we know in advance how many times it will run because we're working with a collection that has a predefined length, like a list.

Here's the general structure:

```
for <temporary variable> in <collection>:
    <task>
```

Let's break it down:
- for: The keyword that starts the loop.
- Temporary Variable: Represents the current element in the collection.
- in: Separates the temporary variable from the collection.
- Task: The list (or iterable) we're looping through.
- Action: The code that runs during each iteration of the loop.

Imagine we have a recipe and want to print each ingredient:

```
ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]

for ingredient in ingredients:
    print(ingredient)

Output:
milk
sugar
vanilla extract
dough
chocolate
```

Explanation:
- Temporary Variable: `ingredient` represents the current item in `ingredients`.
- Collection: `ingredients` is the list we're iterating through.
- Task: `print(ingredient)` prints each item.

Customizing Temporary Variables

The temporary variable's name is arbitrary. Both examples below do the same thing:

```
for i in ingredients:
    print(i)

for item in ingredients:
    print(item)
```

However, descriptive variable names like `ingredient` improve code readability.

Indentation Matters

All actions inside a for loop must be indented:

```
for ingredient in ingredients:
    # Indented code is part of the loop
    print(ingredient)

# Unindented code is outside the loop
print("Done!")
```

2.6.3. For Loops: Using range() for Iterations

Sometimes, we want to repeat an action a certain number of times without iterating through a specific collection. In these cases, we can use the `range()` function with a for loop.

The `range()` function generates a sequence of numbers, which the for loop can iterate over. If we want to perform an action six times, we can use:

```
for <temporary variable> in range(6):
    <task>
```

Suppose you want to track each jumping jack during a workout:

```
for rep in range(5):
    print("Jumping Jack number:", rep + 1)

Output:
Jumping Jack number: 1
Jumping Jack number: 2
Jumping Jack number: 3
Jumping Jack number: 4
Jumping Jack number: 5
```

Explanation:
- `range(5)` generates numbers from 0 to 4.
- Adding 1 to rep displays the count starting at 1.

Using the Temporary Variable Creatively

The temporary variable can be used to make your loop dynamic. Let's create a simple pattern:

```
for step in range(4):
    print("*" * (step + 1))

Output:
*
**
***
****
```

Here:
- step controls how many asterisks are printed during each iteration.

Example: If you're calling on five students in order, you can do this:

```
for number in range(1, 6):
    print("Calling student number:", number)

Output:
Calling student number: 1
Calling student number: 2
Calling student number: 3
Calling student number: 4
Calling student number: 5
```

2.6.4. While Loops: Introduction

In Python, `while` loops are another powerful way to perform repetitive tasks. Unlike `for` loops, which are definite (with a fixed number of iterations), `while` loops are a form of indefinite iteration. They run as long as a specified condition remains true.

The general structure of a while loop is:

```
while <conditional statement>:
    <action>
```

- `while` : Starts the loop.
- `<conditional statement>` : The condition is checked before every iteration.
- `<action>` : The code block that executes if the condition is True.

Here's how you can print the numbers 0 through 3:

```
count = 0
while count <= 3:
    print(count)
    count += 1
Output:
0
1
2
3
```

1. Initialization:

- The variable count is initialized to 0.

2. Condition:

- The condition count <= 3 is evaluated before each iteration.
- If it's True, the loop executes; if False, the loop ends.

3. Action:

- Inside the loop, count is printed, and then it's incremented by 1.

4. Stopping:

- The loop stops when count becomes 4 because count <= 3 is no longer True.

2.6.5. Infinite Loops: What They Are and How to Avoid Them

`While` loops and `for` loops are powerful tools for iteration, they can sometimes lead to infinite loops—loops that never stop running. These loops can freeze your program and consume all available resources.

Consider this code:

```
my_favorite_numbers = [4, 8, 15, 16, 42]

for number in my_favorite_numbers:
    my_favorite_numbers.append(1)
```

What happens?

- During each iteration, 1 is added to the end of the list.
- Since the list keeps growing, the loop never reaches the end.
- The loop runs indefinitely, creating an infinite loop.

Why Are Infinite Loops Dangerous?

1. Resource Consumption:

- Infinite loops continue to run, using CPU and memory endlessly.

2. Unresponsive Programs:

- Your program may freeze or crash, requiring manual intervention.

3. Difficult Debugging:

- Infinite loops can be tricky to detect, especially in complex code.

How to Avoid Infinite Loops

1. Plan Your Conditions:

- Ensure the loop condition will eventually become False.

Example (good condition):

```
count = 0
while count < 5:
    print(count)
    count += 1
```

2. Avoid Modifying the Collection:

- Do not change the size of the collection you're iterating over.

Example (fix the issue above):

```
my_favorite_numbers = [4, 8, 15, 16, 42]

for number in my_favorite_numbers[:]:  # Use a copy of the list
    my_favorite_numbers.append(1)
```

3. Test Your Loops:

- Print debug messages or use counters to confirm that your loop behaves as expected.

If you accidentally create an infinite loop:
1. In Your Code Editor:

- Use Ctrl + C (Windows/Linux) or Command + C (Mac) to terminate the program.

2. In a Web-Based Editor:

- Refresh the page to stop the execution.

2.6.6. Loop Control: The break Statement

Python's `break` statement allows you to terminate a loop prematurely. This can save time and resources, especially when searching through large collections.

Imagine searching for a specific item in a list. Once you find the item, there's no need to continue iterating through the rest of the list. The `break` statement helps stop the loop as soon as the target is found.

Suppose you have a list of books:

```
books = ["The Catcher in the Rye", "To Kill a Mockingbird", "1984", "The Great
Gatsby", "Moby Dick"]
```

You're looking for "1984". Without using `break`, the loop would continue even after finding the book:

```
for book in books:
    if book == "1984":
        print("Found it!")
```

While this works, the loop doesn't stop once "1984" is found. Here's how you can use `break` to end the loop as soon as the target is found:

```
print("Searching for your book...")

for book in books:
    print(f"Checking: {book}")
    if book == "1984":
        print("Found it!")
        break

print("Search complete.")

Output:

Searching for your book...
Checking: The Catcher in the Rye
Checking: To Kill a Mockingbird
Checking: 1984
Found it!
Search complete.
```

Breaking It Down:
1. Iterate Through the List:

- The loop goes through each book, checking if it matches "1984".

2. Stop When Found:

- Once “1984” is found, the break statement is executed, and the loop ends immediately.

3. Skip Remaining Items:

- The loop doesn’t check “The Great Gatsby” or “Moby Dick”.

2.6.7. Loop Control: The continue Statement (7 minutes)

While the `break` statement stops a loop entirely, the `continue` statement skips the current iteration and moves directly to the next one. This is useful when you want to exclude specific cases without terminating the loop.

Imagine processing a list but skipping certain elements based on a condition. The `continue` statement allows you to bypass specific iterations while continuing with the rest of the loop.

Suppose we have a list of numbers:

```
numbers = [1, 2, -1, 4, -5, 5, 2, -9]
```

We want to print only the positive numbers:

```
for num in numbers:
    if num <= 0:
        continue
    print(num)

Output:
1
2
4
5
2
```

Breaking It Down:

1. Check Condition:

```
if num <= 0:
```

If the number is negative or zero, skip it.

2. Skip with continue:

```
continue
```

When the loop encounters continue, it skips the rest of the code in the loop body for that iteration.

3. Process Remaining Numbers:

- Only positive numbers are printed because the loop skips printing for negative numbers.

2.6.8. Nested Loops: Working with Sub-Lists

In Python, you can nest loops to handle more complex data structures, such as lists of lists. Nested loops are especially useful when working with grids, matrices, or groups of sub-lists.

Suppose we're managing a science class divided into three project teams:

```
project_teams = [["Ava", "Samantha", "James"], ["Lucille", "Zed"], ["Edgar",
"Gabriel"]]
```

With a single loop, we can iterate through the sub-lists:

```
for team in project_teams:
    print(team)
Output:
['Ava', 'Samantha', 'James']
['Lucille', 'Zed']
['Edgar', 'Gabriel']
```

This gives us the sub-lists representing each team but doesn't access individual students.

Using Nested Loops for Individual Access

To access each student, we need to add a nested loop:

```
for team in project_teams:
    for student in team:
        print(student)
Output:
Edit
Ava
Samantha
James
Lucille
Zed
Edgar
Gabriel
```

How It Works:

1. Outer Loop:

- Iterates over the main list (project_teams).
- Each iteration processes one sub-list (team).

2. Inner Loop:

- Iterates over each sub-list (team).
- Each iteration processes one item (student) in the sub-list.

2.6.9. Introduction to Python Dictionaries

A dictionary in Python is a collection of key-value pairs. It allows us to associate data so that we can quickly find the values corresponding to specific keys.

A dictionary provides a way to map data together. For example, let's say you're managing a classroom and want to store each student's test score:

```
scores = {"Alice": 95, "Bob": 88, "Charlie": 92}
```

Here's how the dictionary works:
- Keys: "Alice", "Bob", "Charlie" (the names).
- Values: 95, 88, 92 (the test scores).

Dictionary Structure

1. Curly Braces:

- A dictionary starts and ends with { }.

2. Key-Value Pairs:

- Each item is a pair: a key and its associated value, separated by a colon `:`.
- Example: “Alice”: 95.

3. Comma-Separated Items:

- Each key-value pair is separated by a comma.

Imagine keeping track of classroom supplies and their quantities:

```
supplies = {"pencils": 30, "notebooks": 15, "erasers": 20}
```

2.6.10. Creating a Dictionary in Python

Dictionaries in Python allow you to map keys to values. Both keys and values can be of any data type, making dictionaries highly flexible for storing complex data.

You can use numbers as dictionary keys. For instance, to store the cube of specific numbers:

```
number_cubes = {1: 1, 2: 8, 3: 27, 4: 64}
```

Here:
- Keys: 1, 2, 3, 4 (the numbers).
- Values: 1, 8, 27, 64 (their cubes).

Dictionary values can be of any type, including strings, numbers, lists, or even other dictionaries.

Example: Courses and Enrolled Students

```
courses = {
    "math": ["Alice", "Bob", "Charlie"],
    "science": ["David", "Eve", "Frank"],
    "history": ["Grace", "Hannah"]
}
```

- Keys: “math”, “science”, “history” (course names).
- Values: Lists of students enrolled in each course.

Mixing Key and Value Types

Dictionaries allow you to mix and match key and value types in the same dictionary.

Example: Device Information

```
device = {
    "model": "X200",
    "price": 799.99,
    "features": ["5G", "128GB Storage", "Waterproof"],
    "in_stock": True
}
```

- Key `model` maps to a string.
- Key `price` maps to a float.
- Key `features` maps to a list.
- Key `in_stock` maps to a Boolean.

2.6.11. Invalid Keys in Dictionaries

In Python, while dictionary values can be of any type, keys must be immutable and hashable. This means certain data types, like lists and dictionaries, cannot be used as keys.

Lists are mutable, meaning their contents can be changed after they are created. Python requires dictionary keys to be immutable to ensure consistent behavior when looking up values. If you try to use a list as a key, you'll get a `TypeError`.

The following code attempts to use lists as keys:

```
powers = {[1, 2, 4, 8, 16]: 2, [1, 3, 9, 27, 81]: 3}

TypeError: unhashable type: 'list'
```

Explanation:
- The term "unhashable" means the object (a list, in this case) cannot be used as a key because it can be modified.

Valid Alternatives

To fix this, use an immutable type like a tuple instead of a list:

```
powers = {(1, 2, 4, 8, 16): 2, (1, 3, 9, 27, 81): 3}
```

Here:
- Keys: (1, 2, 4, 8, 16) and (1, 3, 9, 27, 81) (tuples).
- Values: 2 and 3.

2.6.12. Adding a Key-Value Pair to a Dictionary

In Python, you can easily add a new key-value pair to a dictionary using the syntax:

```
dictionary[key] = value
```

Suppose you have a menu dictionary:

```
menu = {
    "oatmeal": 3,
    "avocado toast": 6,
    "carrot juice": 5,
    "blueberry muffin": 2
}
```

To add a new item, "cheesecake" priced at 8, you would write:

```
menu["cheesecake"] = 8
```

Now, the dictionary looks like this:

```
{
    "oatmeal": 3,
    "avocado toast": 6,
    "carrot juice": 5,
    "blueberry muffin": 2,
    "cheesecake": 8
}
```

Overwrite Values

We know that we can add a key by using the following syntax:

```
menu["banana"] = 3
```

This will create a key "banana" and set its value to 3. But what if we used a key that already has an entry in the menu dictionary?

In that case, our value assignment would overwrite the existing value attached to that key. We can overwrite the value of "oatmeal" like this:

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
menu["oatmeal"] = 5
print(menu)

Output:
{"oatmeal": 5, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
```

2.6.13. Dictionary Comprehensions

Python offers a concise way to create dictionaries by combining lists or other iterable objects using dictionary comprehensions. This method is efficient and clean, particularly when transforming or pairing data from multiple sources.

Suppose we have a list of students and their heights:

```
names = ["Jenny", "Alexus", "Sam", "Grace"]
heights = [61, 70, 67, 64]
```

To combine these into a dictionary where the names are the keys and the heights are the values, we can use a dictionary comprehension:

```
students = {key: value for key, value in zip(names, heights)}
print(students)

Output:
{'Jenny': 61, 'Alexus': 70, 'Sam': 67, 'Grace': 64}
```

How It Works:

1. `zip()`:

- Combines names and heights into pairs: `[("Jenny", 61), ("Alexus", 70), ...]`.

2. Iteration:

- The dictionary comprehension iterates through these pairs.

3. Key-Value Mapping:

- Each pair is split into key (from names) and value (from heights).

4. Dictionary Creation:

- Each key: value pair is added to the dictionary.