Python

Morteza Ghodoousi

Writing Code

Python is one of the most popular and useful languages used to give instructions to computers.

In this course, you'll learn to write programs to automate and speed up tasks.

Writing Code

Python has a simple syntax which means it's easy to write, read and learn!

print("Welcome")

The print() instruction is the easiest way to send a message to the screen or other display device.

print("Level Up!")

The print() instruction requires the use of parentheses around the message.

Make sure you use quotation marks around the text messages.

If you miss the quotation marks around the text the computer won't understand your instructions.

Numbers don't require quotation marks.

Writing Code

Lesson Takeaways

Amazing job! Python is widely used to build software and games, analyze data and program Artificial Intelligence.

You learned that:

- humans use code to give instructions to machines
- the print() instruction displays a message on the screen

In the next lesson, you'll learn to write code to store important information.

Memory & Variables

Computer programs use variables to remember important information, like items in a shopping cart, prices and discounts.

The line of code below tells the computer to store information in a variable called item.

```
item = "bike"
```

To create a variable, you just need to give it a name.

The information that you need to store is added on the right.

Variables have a name and a value. They are connected with the equal sign =.

You can think of a variable as a box that contains some information.

The line of code below tells the computer to store information in memory.

```
message = "Level Up"
```

Memory & Variables

Lesson Takeaways

Amazing job! You learned that:

- Computer programs use variables to remember important information
- A variable has a name and a value
- You can create a variable by connecting the name and the value with an equal sign =

In the next lesson, you'll start writing and running real code.

Text Data

A large amount of information out there consists of text. A piece of text data is called a string.

In this lesson, you'll work with text data.

Text Data

Strings in Python need to be surrounded by quotation marks.

We use quotation marks to tell Python that we are working with a piece of text data.

Strings can be stored in variables.

In Python, both single ' and double " quotes can be used to define strings.

It doesn’t make a difference whether you choose double or single quotation marks.

The quotation marks just need to match.

Text Data

A computer program is made of lines of code.

You can add as many lines and variables to your code as you need to give the computer instructions.

The code in computer programs is made of statements.

Statements are the instructions for the computer to follow. Real programs can contain thousands of statements.

What's the number of statements in this code?

```
book = "Harry Potter"
author = "J. K. Rowling"
```

The print() statement is the easiest way to send a value to the screen.

```
print("Iceland")
```

Text Data

Lesson Takeaways

Well done! You learned that:

- A piece of text is called a string
- Strings require quotation marks
- The print() statement is used to send a value to the screen

In the next lesson, you'll learn about other types of data.

Numerical Data

Numerical data is information that comes in the form of numbers.

In this lesson, you'll learn how to deal with numerical data in your computer programs.

Numerical Data

Numerical values can be directly stored in variables.

Numerical data shouldn't be in quotation marks. The line of code below declares a numerical variable

```
points = 500
```

You can send a number to the screen with the print() statement.

You just need to insert the number between the parentheses.

You can perform math operations with numbers. Each print() instruction will add a value to the screen in a new line.

```
print(7 + 3)
print(10 - 5)
print(5 * 3)
print(10 / 2)
```

Numerical Data

You can use the print() statement to check that the computer is following your instructions.

A variable's name is used to identify where that information is stored.

You can access the value that a variable is storing by calling its name.

When run, what will this code show on the screen?

```
budget = 200
print(budget)
```

The fragment below is part of a prototype for a new video game. Which 2 values will this code display on the screen?

```
username = "magician"
points = 50
lives = 3
print(username)
print(points)
```

Numerical Data

Lesson Takeaways

Great work! You completed the lesson. You learned that:

- Numerical values can be stored in variables
- You can access the value stored in a variable by calling its name
- Numerical data should not be surrounded by quotation marks

In the next lesson, you'll learn to work with the data you stored in variables.

Task

Complete the code to display the "Game over" message on the screen

Working with Variables

Variables are key to software development. They allow you to store, label and play with data.

In this lesson, you’ll learn to work with data that has been stored in variables.

Working with Variables

You can access the value stored in a variable by calling its name.

What will this code send to the screen?

```
price = 150
print(price)
```

You can make calculations using the values in variables. What will this code send to the screen?

```
budget = 20
print(budget + 10)
```

What will this code send to the screen?

```
price = 5
amount = 3
print(price * amount)
```

Working with Variables

You can store the result of a calculation in a variable.

What's this code sending to the screen?

```
score = 7 + 8
print(score)
```

You can create a new variable to store the result of a calculation made using other variables.

You can update the value stored in a variable. The variable will forget the previously stored value.

```
price = 99
price = 100
print(price)
```

Working with Variables

Updating the value of a variable is called reassigning a variable.

```
points = 35
points = 45
print(points)
```

What values will the code send to the screen?

```
name = "Tom"
level = 14
print(name)
level = level + 1
print(level)
```

Working with Variables

Lesson Takeaways

Great job! You learned that:

- You can run calculations using the values stored in variables
- You can store the result of a calculation in a variable
- Updating the value of a variable is called reassigning a variable

In the next lesson, you'll start fixing errors in broken code.

Task

- Create a variable to store the numerical value and display it on the screen
- Complete the code that calculates the semester grade by adding the midterm and final exam grades together, and then displays the result on the screen

Expected Result

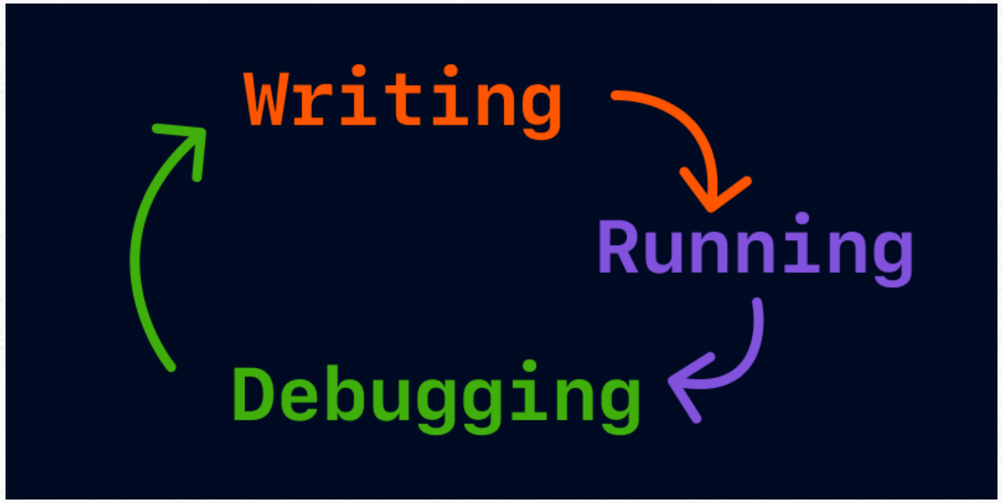
Sum of midterm and final exam 77+23 = 100

Debugging

Coding consists of 3 steps:

- Writing
- Executing (or running)
- Fixing errors (or debugging)

In this lesson, you'll start step 3: identifying and fixing errors.



Debugging

Machines will refuse to follow instructions from a human if they contain mistakes.

The code below contains an error. Python will return an error message. "Run" the code to get the error message.

```
message = "Debugging

print(message)
```

Fix the error by adding the missing closing quotation mark, then "Run" again.

Even experienced programmers get coding errors (or bugs) all the time.

So if you're making mistakes, it's ok! You're on the right track.

Debugging

The line of code below contains a bug, can you identify it?

```
print 5 + 3
```

Even the tiniest typo or a misplaced tab in your code will result in an error.

The line of code below is meant to create a new variable but something is wrong

```
color -> "red"
```

Select the name of the variable that is storing a string

```
surname = "Smith"
age = 32
```

Debugging

Bugs are an expected part of writing code. The trick to being a good coder is staying cool when dealing with bugs.

A piece of code can contain multiple bugs. Can you fix the code to display the string?

```
print"Great Progress)
```

If your code contains multiple bugs, you need to fix all of them for the code to be executed correctly.

Which lines contain bugs?

```
salary = 50000
role = Analyst
age -> 29
print(salary)
```

Debugging

The computer reads and executes instructions line by line, from top to bottom.

The execution of the program will be interrupted at the first error encountered.

At which line will this code stop the execution?

```
salary = 50000
role = Analyst
age -> 29
print(salary)
```

The code below contains bugs in lines 2 and 3.

The computer will get stuck at line 2. Why?

After an error is encountered, the computer will stop reading and executing code.

Debugging

At which line will this code stop execution?

```
salary = 70000
print(salary)
age -> 32
role = "Product Manager"
print(role)
```

Calling the name of a variable that has not been defined is a very common error.

What will this code send to the screen?

```
print(balance)
```

The code below will return an error. Why?

```
print(revenue)
revenue = 300
```

Debugging

Lesson Takeaways

You did it! You learned that:

- Errors in code are known as bugs
- Code is executed line by line from top to bottom
- Code execution is interrupted by bugs

In the next lesson, you'll start applying the standards and best practices that professional coders use.

Standards and Best Practices

Millions of coders develop and maintain computer programs every day.

Universal standards and best practices are needed for collaboration.

In this lesson you'll start applying some of these standards and best practices used in the tech industry.

Standards and Best Practices

Professional developers use comments to add descriptions and explanations to their programs.

You can add comments to your Python code with the hash symbol #

Comments make code easier to read for humans.

Comments will help others (and future you!) to understand the code.

Comments are not instructions, they are ignored by machines.

```
#Declaring payment variables
currency = "USD"
amount = 200
status = "overdue"
#Displaying status
print(status)
```

Standards and Best Practices

You can use comments to temporarily disable a statement. This way the computer will skip the instruction.

What will this code send to the screen?

```
#print("Game Over")
```

This code will result in an error. Why?

```
# client = "loyal"
print(client)
```

Standards and Best Practices

Python is a case-sensitive language, meaning "A" and "a" are treated as different.

How many different variables are being declared in the code below?

credit = 300

Credit = 280

CREDIT = 320

Python is a case-sensitive language.

True or False? You can get errors if you don't pay attention to the use of uppercase and lowercase.

Standards and Best Practices

Snake case is a popular way to create variable names in a consistent way.

Snake case uses underscores _ to separate words in a variable name.

Coders use snake case to give descriptive names to variables with multiple words.

The underscore makes the variable name easier to read.

Standards and Best Practices

Lesson Takeaways

Great work! You learned that:

- You can add comments to your code with the hash symbol #
- Python is a case-sensitive language
- Snake case is the best practice when creating multi-word variable names

In the next lesson, you'll apply these best practices to avoid bugs.

Applying Best Practices

It's time to put into practice the skills and best practices you've learned.

You'll avoid bugs and solve problems involving variables and different data types.

Spaces are not allowed in variable names. Python will return an error if your variable names contain spaces.

Remember that Python is a case-sensitive language.

A variable name can contain numbers but cannot start with a number.

You can use a variable to construct a new variable.

Applying Best Practices

What will this code send to the screen?

```
salary = 900

new_salary = salary + 200

print(new_salary)
```

What will this code send to the screen?

```
salary = 1000

pay_raise = 100

new_salary = salary + pay_raise

print(new_salary)
```

Applying Best Practices

What will this code send to the screen?

```
salary = 1000
pay_raise = 100
print(new_salary)
new_salary = salary + pay_raise
```

What will this code send to the screen?

```
salary = 1000
pay_raise = 100
new_salary = salary + pay_raise
print(salary)
```

Applying Best Practices

What will this code display on the screen?

```
payment_status = "returned"

payment_status = "paid"

print(payment_status)
```

What will this code send to the screen?

```
a = 3

a = 5

a = 7

print(a)
```

Applying Best Practices

Lesson Takeaways

You are unstoppable! You learned that:

- Spaces are not allowed in variable names
- A variable name cannot start with a number
- Best practices can help you avoid errors

In the next lesson, you'll learn how to make your programs interactive.

Inputs and Outputs

Computer programs are designed to interact with users and the outside world.

In this lesson, you'll learn to create code that takes information in and sends information out.

Inputs and Outputs

An input is any information that goes into a computer.

The press of a key and the click of a button are examples of inputs.

The input() instruction is the easiest way to allow a user to insert a value into your program.

Run the code to give it a try

```
message = input()

print(message)
```

Inputs and Outputs

An output is a way for the computer to communicate with the outside world.

A message displayed on the screen and the sound from a speaker are examples of outputs.

The print() instruction, which you already know, is the easiest way to get your computer program to generate an output.

The code below contains a bug. Can you fix it to generate the output?

```
print 360
```

Inputs and Outputs

A robot vacuum cleaner makes a beeping sound when it's stuck. This sound is an example of an output.

A robot vacuum cleaner is equipped with a sensor that helps it navigate its environment and detect obstacles.

The information coming from the sensor is an example of an input.

Inputs and Outputs

What's the value stored in the proximity_sensor variable after the updates?

```
proximity_sensor = "obstacle"

proximity_sensor = "clear"
```

Computer programs can have multiple inputs and outputs. What will the code output?

```
username = "zombie"

points = 50

lives = 3

print(points)

print(lives)
```

Inputs and Outputs

Lesson Takeaways

You did it! You learned that:

- inputs and outputs help machines communicate with the outside world
- the input() instruction allows the user to enter a value into your program
- the print() instruction is used to generate an output

In the next lesson, you'll learn how to handle different types of data.

Data Types

Data comes in different shapes and forms. Computers treat different types of data in different ways.

In this lesson, you'll start working with different types of data.

Data Types

String is the data type for a piece of text.

The quotation marks tell the computer that a value needs to be stored as a string.

Computers store and process different types of numbers.

Integers are whole numbers without a decimal point. They can be positive, negative or zero.

Float is the data type for numbers with decimal places, they can be positive or negative.

What's this variable storing?

variable = 5/2

You have been introduced to 3 different data types.

Data Types

The way computers operate with values depends on the data type.

Run the code to see the output

```
print(3 + 5)
print("Iron" + "Man")
```

When you use the + addition operator with string values the two strings are joined together.

This is known as concatenation.

Data Types

What will the code output?

```
a = "basket"

b = "ball"

print(a+b)
```

Anything in quotation marks will be treated as a string, even numbers.

What's the value stored in the variable?

```
var = "360"
```

You won't be able to do math operations if numbers are surrounded by quotes. They will be treated as strings.

What would be the output?

```
print("360" + "360")
```

Data Types

Lesson Takeaways

Great job! You learned that:

- computers store and process different data types differently
- string is the data type for text
- integer and float are data types for numbers

In the next lesson, you'll learn how to detect data quality issues.

Data Type Checking

Data comes in a variety of shapes and forms.

Dealing with data in the incorrect format can result in data loss or corruption.

In this lesson, you'll learn to check the data type stored in a variable.

Data Type Checking

What's the output of this code?

```
a = "note"

b = "book"

print(a+b)
```

The code below will result in an error. You can't add a number to a string.

Run the code to get the error message

```
a = 3

b = "8"

print(a+b)
```

Can you fix the code to output the number 11?

Data Type Checking

Data can come to you in the incorrect format.

You can use the type() instruction to check the data type stored in a variable.

```
balance = "780"
type(balance)
```

Run the code to output the data type stored in the variables

```
city = "Berlin"
age = 42
balance = 830.29
```

Data Type Checking

What will this code output?

```
balance = 234.3
print(type(balance))
```

The code will return an error. Why?

```
budget = 200
expenses = "180"
savings = budget - expenses
```

Data Type Checking

The division of two integers always produces a float.

a = 4

b = 2

c = 4/2

What will this code output?

x = 9

y = 3

print(x/y)

Data Type Checking

Lesson Takeaways

Great job! You learned that:

- the type() instruction is used to check the data type
- the division of two integers always produces a float

In the next lesson, you'll learn how to convert data from one type to another.

Data Conversion

Data can come in the incorrect format. Data from surveys and web forms can come to you with quality issues.

In this lesson, you'll learn to convert data to get it into the correct format.

Data Conversion

The input() instruction always turns the user input into a string, no matter what the user enters.

Run the code to check the data type

```
birth_year = input()
print(type(birth_year))
```

What's this variable storing after the user enters a number?

```
project_budget = input()
```

Data Conversion

You can convert data from one type to another to fix data quality issues.

The int() instruction converts any type of value into an integer

x = "55" #x is a string

y = int(x) #y is an integer

You can use the int() instruction to convert the user input into an integer

height = int(input())

Match the variable with the data type that will store when the user enters a number

a = input()

b = int(input())

Data Conversion

There are situations when you need values to be treated as floats.

The float() instruction converts values into floats.

What will the code output?

```
a = 3
b = float(a)
print(b)
```

In a similar way, you can ensure that values are converted into strings with the str() instruction.

The int(), str() and float() instructions are examples of explicit conversion,

which means they are performed by an instruction given by a programmer (like you).

Data Conversion

On the other hand, run the code to see some examples of implicit (automatic) data type conversions

```
x = 5 # integer
y = 2 # integer
z = x/y # float (implicit conversion)
```

What will the code output?

```
x = input()
y = input()
print(x+y)
```

Data Conversion

Lesson Takeaways

Great job! You learned that:

- you can change the data type of a value with int(), float() and str()
- there are implicit and explicit data type conversions in Python
- str(), int(), float() instructions are explicit conversions

In the next lesson, you'll fix more issues with data types.

Fixing Data Types

The execution of your program can fail if your data is in the incorrect format.

In this lesson, you’ll put your data type conversion skills into practice to fix data quality issues.

Fixing Data Types

You need to ensure that the score entered by the user is stored as an integer.

What will this code output after getting a number from the user?

```
age = int(input())
print(type(age))
```

The code will output <class 'int'>

```
x = 15
print(type(x))
```

This code for a navigation app will result in an error. Why?

```
distance = 14
units = "km"
print(distance + units)
```

What will this code output?

```
print(14 + "km")
```

Fixing Data Types

What will the code output?

```
print("14" + "km")
```

The str() command can help you with concatenations.

The code contains multiple errors. Which lines contain errors?

```
investment = 120000

rate -> 0.1

print investment * rate
```

Math operations between integers and floats produce a float.

Fixing Data Types

What will the code display on the screen?

```
x = 9

y = 3.0

print(x+y)
```

This code will display a float.

```
print(12/6)
```

Fixing Data Types

Lesson Takeaways

Great job! You learned that:

- you can use explicit data type conversions to avoid bugs in your programs
- int() ensures that the user input is treated as an integer number
- str() can help you concatenate numbers with text

In the next lesson, you'll learn to get machines to make decisions on their own.

Comparison Operations

Computers are faster and more precise than humans at certain operations.

In this lesson, you'll learn about a type of operation that makes machines evaluate different scenarios and make decisions.

Comparison Operations

Comparison operations are key to the development of computer programs. The line of code below shows an example of a comparison operation.

```
5 < 9
```

A comparison operation always results in either one of these two outcomes: Yes or No

```
50 > 100
```

Both statements will instruct the computer to perform the comparison operation.

Comparison Operations

Run the code to check the result of different comparison operations in Python

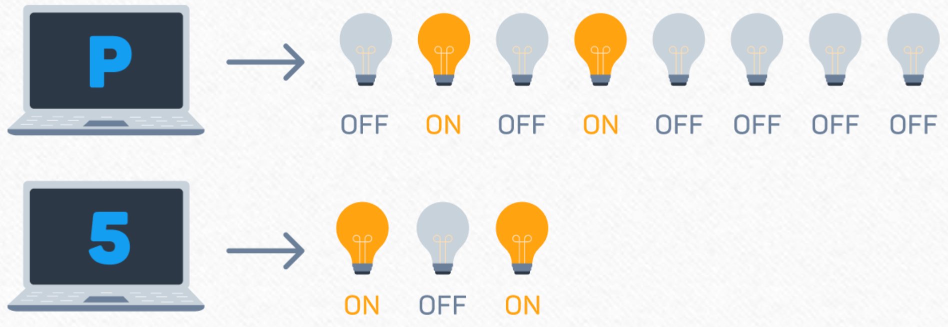
```
print(30 < 25)

print(5 < 9)

print(50 > 100)
```

The result of a comparison operation in Python is either True or False.

Comparison Operations

Electronic circuits inside computers use millions of tiny switches to store these True/False values.

Computers use binary code to represent information. By turning switches ON and OFF, we change the information stored in a computer.



Comparison Operations

To simplify things, two numbers are used to represent the OFF/ON states of a switch.

Complex modern tasks like video streaming or online banking transactions can be broken down into tiny simple calculations.

You are now ready to meet another data type. The Boolean is a data type that only has two possible values: True or False.

Run the code to see the data type of the values

```
print(type(5 < 9))

print(type(50 > 100))
```

Comparison Operations

This data type is named after George Boole, who created the theory that is the basis for modern computers.

Comparison operations and Boolean values allow machines to make decisions.

Let's look at a real-world example. A plant watering system uses a sensor to measure soil moisture.

When the soil moisture is less than 100 units, the system will automatically water the plant. The True value will trigger the action.

You can use Booleans to trigger actions. After the plant has been watered, the reading from the soil moisture sensor will increase.

Comparison Operations

What will the code output?

```
soil_moisture = 80

soil_moisture = 120

print(soil_moisture < 100)
```

Comparison Operations

Lesson Takeaways

Amazing! You learned that:

- the Boolean data type has one of two possible values: True or False
- a comparison operation always results in a Boolean

In the next lesson, you'll learn to build programs that evaluate complex scenarios.

Logical Operations

Modern computers can perform complex tasks very fast because these can be broken down into lots of tiny, simple calculations.

In this lesson, you'll learn about another type of operation that computers can do faster than humans.

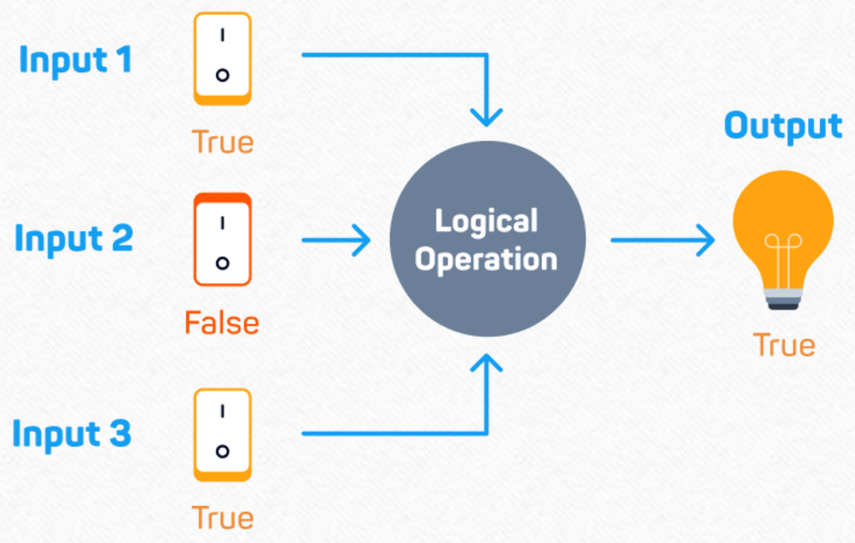
Logical Operations

Logical operations are needed for machines to evaluate complex scenarios.

Logical operations use Boolean values.

A logical operation

- takes several Boolean Inputs
- produces only 1 Boolean Output

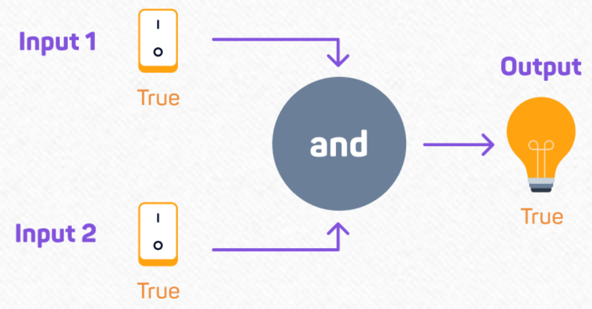


Logical Operations

The "and" operation results in a True value only when all the inputs are True at the same time.

What's the result of this "and" logical operation?

True and True



Logical Operations

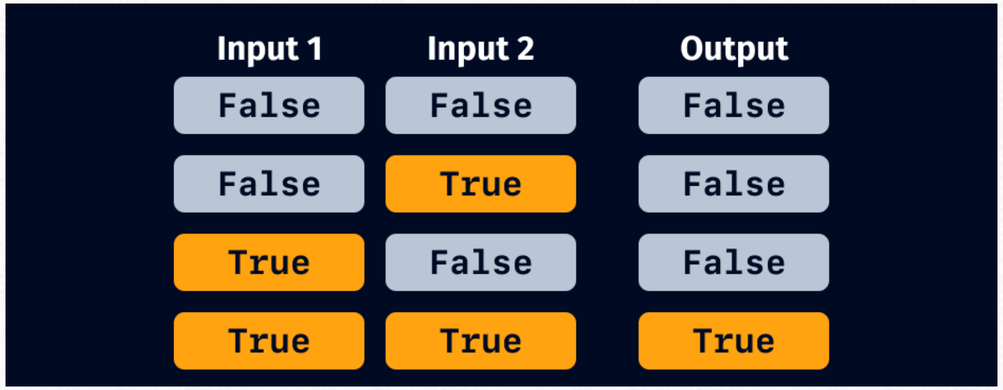
A logical operation combines Boolean inputs to produce a Boolean output.

True and True = True

Every other combination = False

What's the result of the "and" logical operation?

True and False

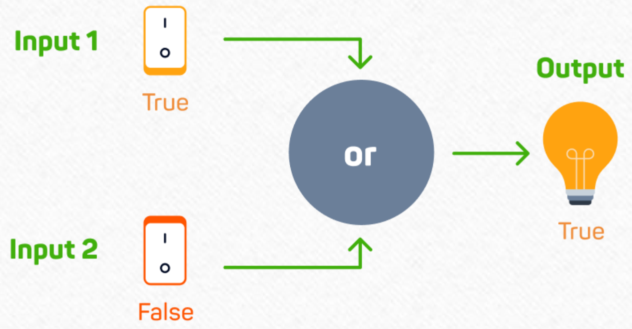


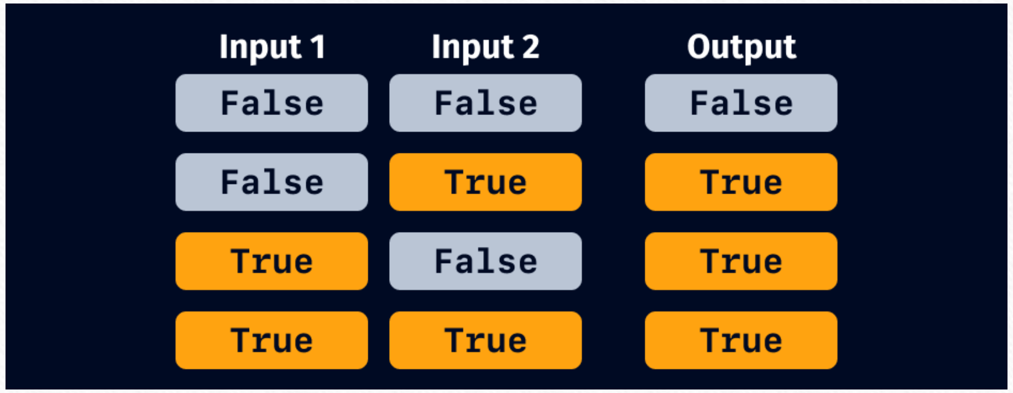
Input 1	Input 2	Output
False	False	False
False	True	False
True	False	False
True	True	True

Logical Operations

The "or" logical operation results in a True value if at least one of the inputs is True. What's the result of this "or" logical operation?

True or False





Input 1	Input 2	Output
False	False	False
False	True	True
True	False	True
True	True	True

Logical Operations

What's the result of this logical operation?

True and True

Logical operations need to be inserted in print() instructions for the result to be outputted.

print(True and False)

print(False and True)

print(True or False)

print(False or True)

Both lines of code below will perform the logical operation.

Logical Operations

Lesson Takeaways

Great work! You learned that:

- logical operations take multiple Boolean values as input
- logical operations produce a single Boolean value as output
- "and" and "or" are examples of logical operations

In the next lesson, you'll combine logical and comparison operations to solve real-world problems.

Combining Comparison and Logical Operations

In this lesson, you'll mix comparison and logical operators to make even fast, more accurate programs

You can store boolean values in variables just like you do with other data types.

What will the code output?

```
heart_rate = 165
peak_zone = heart_rate > 160
print(peak_zone)
```

You can store the result of a logical operation in a variable.

What will this code output?

```
a = True and False
print(a)
```

You can operate with values stored in variables.

Combining Comparison and Logical Operations

What will this code output?

```
light_on = True
door_locked = False
print(light_on or door_locked)
```

Python is a case-sensitive language.

- Both "True" and "False" start with an uppercase letter.
- Both "and" and "or" operators are lowercase in Python.

Fix the errors in the code

```
light_on = true
door_locked = false
print(light_on OR door_locked)
print(light_on AND door_locked)
```

The temperature control system turns on the air-conditioning when the temperature is greater than 30 °C.

Combining Comparison and Logical Operations

What will this code output?

```
temp = 35
ac_on = temp > 30
print(ac_on)
```

You can use parentheses to group operations so they can be evaluated first. The parentheses make the code easier to read.

```
a = (3 > 2) or False
```

When will the following operation result in a True value?

```
(temp > 30) or presence
```

Combining Comparison and Logical Operations

Lesson Takeaways

You did it! You learned that:

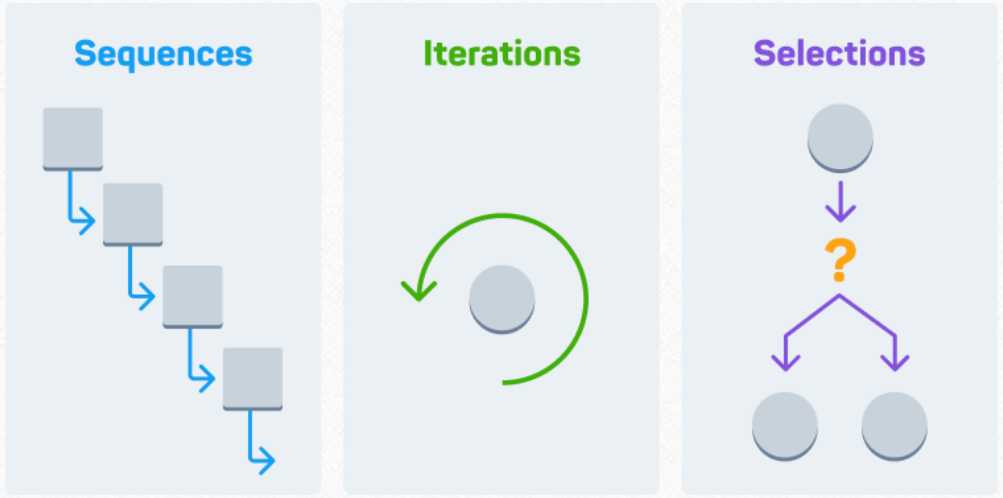
- You can store boolean values in variables
- You can store the result of logical and comparison operations in variables
- You can combine operations with logical and comparison operators

In the next lesson, you'll learn to control the order in which computers execute instructions.

Control Flow

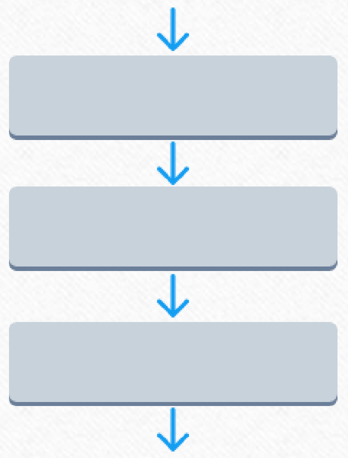
Computers are capable of making decisions, and help us solve real-world problems because they are very good at following instructions.

In this lesson, you'll learn to control the flow of instructions a computer follows, using 3 building blocks: sequencing, iteration and selection.



Control Flow

You already know sequencing. It means that the computer runs your code in order, from top to bottom.



Control Flow

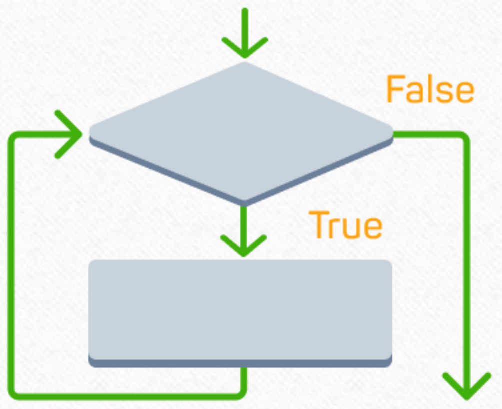
Put the outputs in the order they will be displayed

print("3")

print("building")

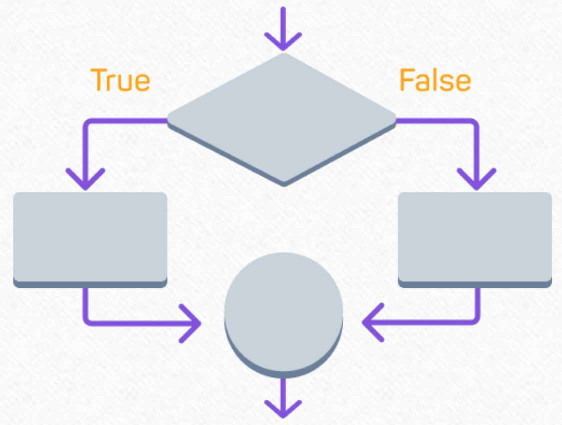
print("blocks")

Iteration is about executing an instruction repeatedly. Iteration is commonly represented as a loop.



Control Flow

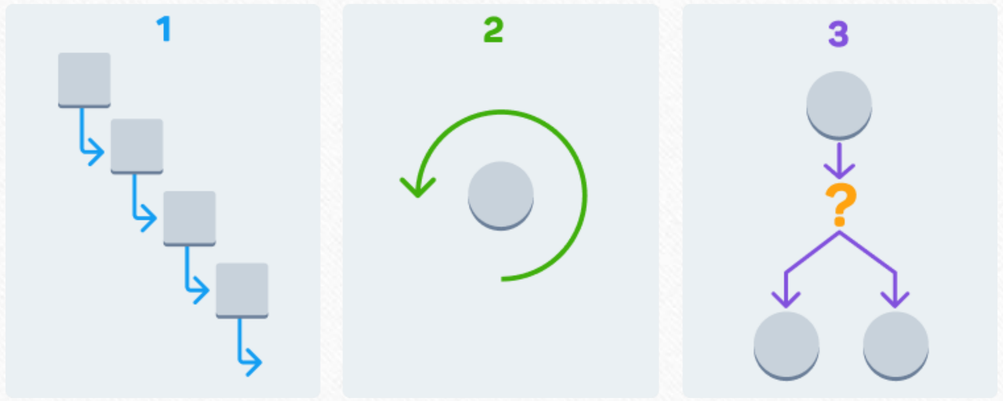
Selection chooses what path a program takes.



Control Flow

Real computer programs perform complex tasks by combining the 3 building blocks.

Match the diagram to the building block



Control Flow

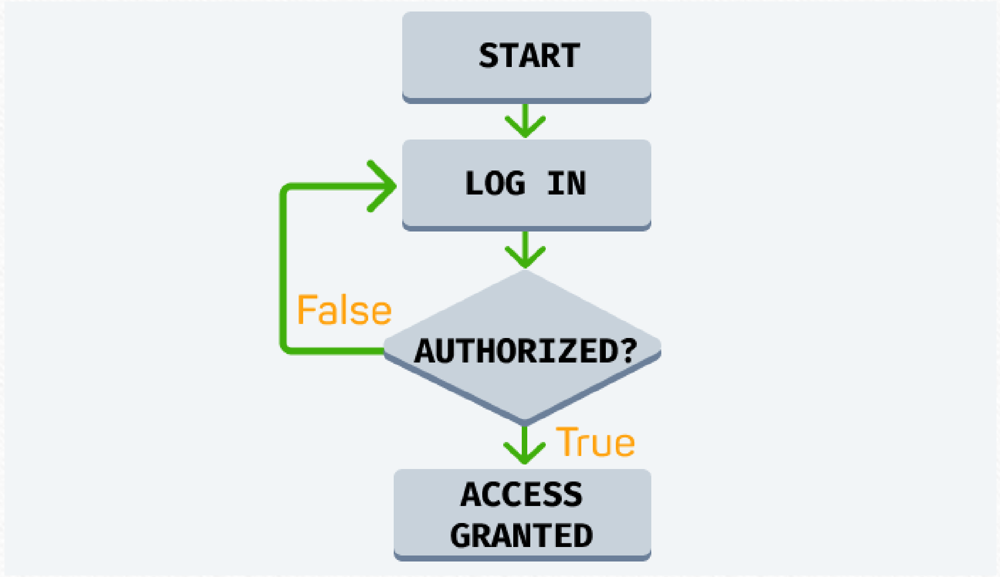
Machines can complete complex tasks for us, but first they need to know how.

An algorithm is a set of step-by-step instructions to complete a task, placed in a certain order.

Algorithms exist in our everyday lives.

Algorithms can be represented in many ways.

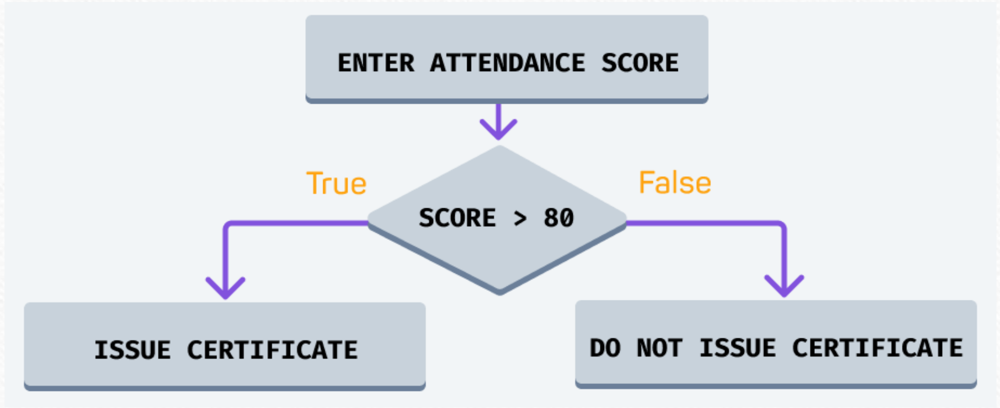
For example, flowcharts help to visualize algorithms.



Control Flow

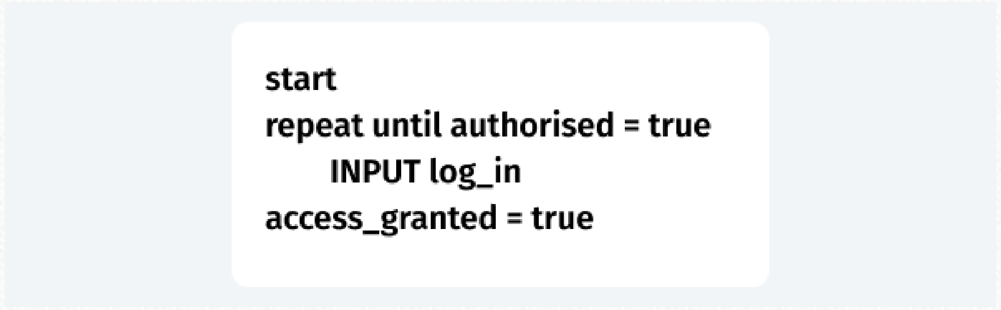
Comparison and logical operations are needed to fully control the flow of your computer programs.

In the diagram, a college awards certificates to students with high attendance scores.



Control Flow

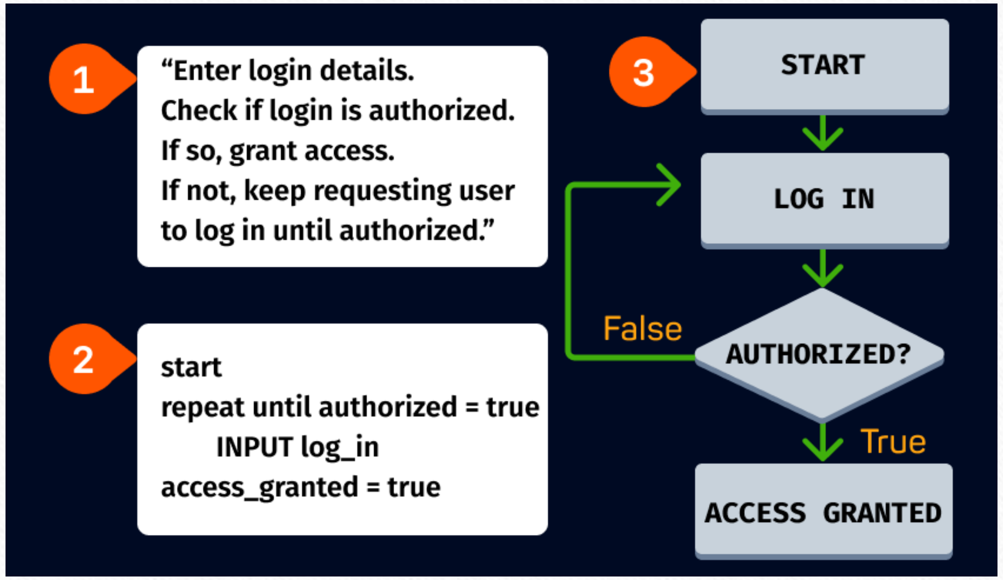
Another way to represent an algorithm is with pseudocode. Pseudocode is a simplified language that is a bit closer to a programming language.



```
start
repeat until authorised = true
    INPUT log_in
access_granted = true
```

Control Flow

Algorithms can be represented in different ways. If you are new to algorithms, natural language is a good place to start.


1

"Enter login details.
Check if login is authorized.
If so, grant access.
If not, keep requesting user to log in until authorized."

2

```
start
repeat until authorized = true
      INPUT log_in
access_granted = true
```

3

Control Flow

Lesson Takeaways

Fantastic job! You learned that:

- you use sequencing, iteration and selection to control the flow of instructions
- an algorithm is a set of step-by-step instructions to complete a task
- algorithms can be represented in different ways

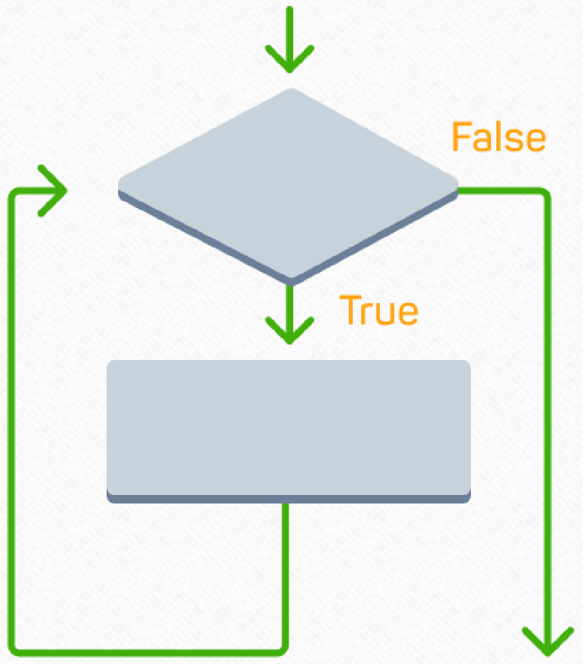
In the next lesson, you'll dive into using iterations to automate tasks.

For Loops

Iteration is used to automate tasks that need to be done over and over again.

Iteration makes your programs simpler, faster and reduces errors.

In this lesson, you'll learn how to create "for loops".



For Loops

A for loop is used to execute the same instruction over and over again, a specific number of times.

Can you make the code print the Hello message 10 times?

```
for i in range(3):
  print("Hello")
```

The for loop begins with the keyword for. The variable i keeps track of the number of iterations.

```
for i in range(100):
  print("Hello")
```

How many times will Hello be outputted?

For Loops

range() creates 10 numbers in a sequence, starting from 0.

```
for i in range(10):
  print(i)
```

range(5) will generate a sequence of 5 numbers.

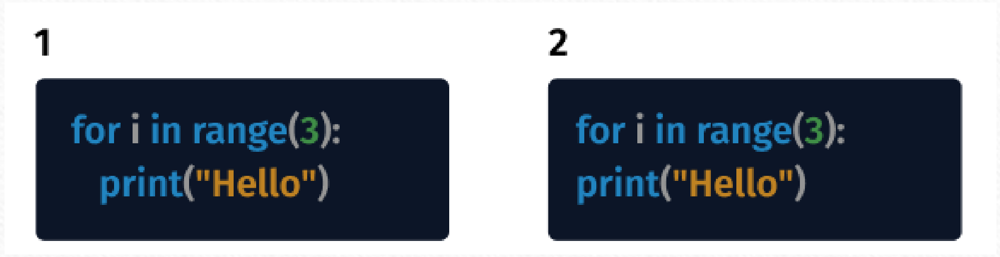
range() generates a series of integer numbers. The i variable is used to iterate over the numbers.

You can replace i with any other variable of your choice.

```
for i in range(3):
  print(i)
```

For Loops

The code that gets repeated in the for loop must be indented. Indentation is the spaces at the beginning of lines.


1

```
for i in range(3):
  print("Hello")
```

2

```
for i in range(3):
print("Hello")
```

For Loops

Code that is not correctly indented will result in errors.

Python doesn't mind whether you use 2 spaces or 4 spaces (or any other number of spaces) as long as you are consistent.

Run the code to get the error message

```
for i in range(3):
print("Hello")
```

Can you fix the bug? In this course we’ll always use 2 spaces.

What gets repeated in the code for the for loop?

```
for i in range(3):
  print("Hello")
```

For Loops

What will the code output?

```
for i in range(5):
print("Congrats!")
```

The code will result in an error. Why?

```
for num in range(5):
print("Coding is fun!")
```

For Loops

The initial loop statement must be followed by a colon : symbol. This signals the start of the iteration block.

This code will result in an error. Why?

```
for num in range(6)
  print(num)
```

For Loops

Lesson Takeaways

Great work! You learned that:

- you can implement iteration into your programs with the for loop
- the initial loop statement must be followed by a colon : symbol
- the code that gets repeated must be indented

In the next lesson, you'll continue to build iteration into your programs with the while loop.

While Loops

While loops are powerful because they can be used even when you don’t know how many iterations will be needed.

How many times does this program output "For Loop"?

```
for i in range(5):
  print("For Loop")
```

While loops repeat code whilst a condition holds true.

For example, a ticket seller at a theater will repeatedly sell tickets until all seats have been occupied

```
while seats > 0:
  print("Sell ticket")
  seats = seats - 1
```

While Loops

While loops begin with the keyword while.

The while keyword is followed by the condition under which the code is repeated

When the condition no longer holds true, we exit the while loop.

```
seats = 300
while seats > 0:
  print("Sell ticket")
  seats = seats - 1
```

When will the loop stop printing the message?

How many times will the “Sell ticket” message be displayed?

```
seats = 300
while seats > 0:
  print("Sell ticket")
  seats = seats - 1
```

While Loops

If the code that gets repeated inside the loop is not indented, the code will result in an error.

Can you fix the bug?

```
while seats > 0:
print("Sell ticket")
  seats = seats - 1
```

As with for loops, the initial while loop statement must be followed by a colon : symbol

Loops usually include counters. A counter is a variable that keeps track of the number of iterations.

```
seats = 300
while seats > 0:
  print("Sell ticket")
  seats = seats - 1
```

What's the counter being used in the tickets example?

While Loops

Counter variables are updated inside the loop, so they change with every iteration.

An initial value is set outside the loop, as the starting point.

What happened to the value of seats with every interaction?

With while loops you can run into what is known as an infinite loop. This is when the condition holds true forever, and the code never stops repeating.

Counters help you avoid infinite loops.

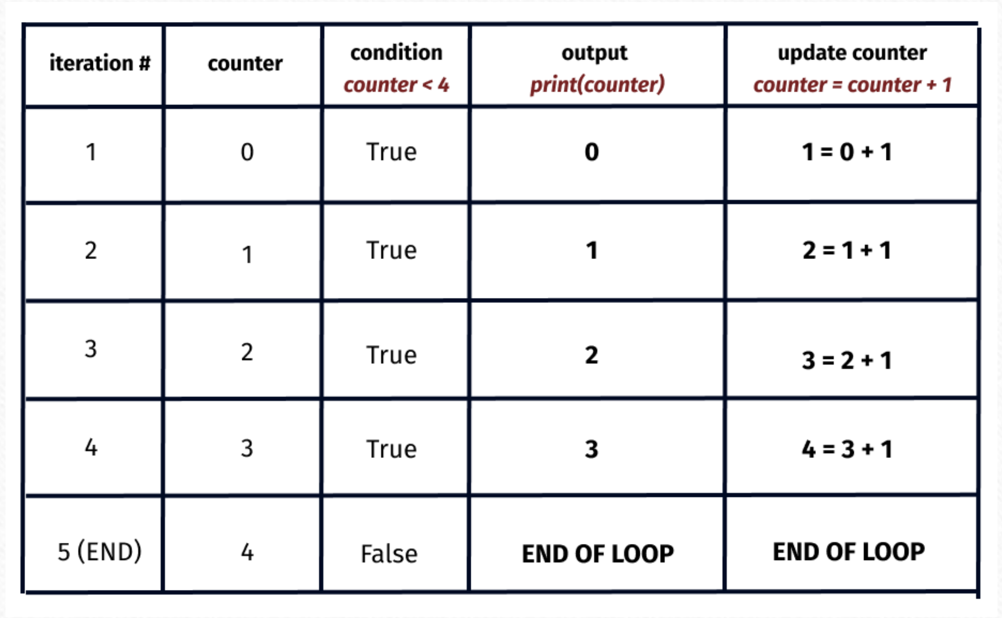
```
seats = 300

while seats > 0:

  print("Sell ticket")

  #seats = seats - 1
```

While Loops

Let's take a closer look at what happens inside a loop

```
counter = 0
while counter < 4:
  print(counter)
  counter = counter + 1
```

In general, use for loops when you already know the number of iterations, and while loops when there is a condition that needs to be met.



iteration #	counter	condition *counter < 4*	output *print(counter)*	update counter *counter = counter + 1*
1	0	True	**0**	**1 = 0 + 1**
2	1	True	**1**	**2 = 1 + 1**
3	2	True	**2**	**3 = 2 + 1**
4	3	True	**3**	**4 = 3 + 1**
5 (END)	4	False	**END OF LOOP**	**END OF LOOP**

While Loops

Lesson Takeaways

Fantastic work! You learned that:

- you can apply iteration to your programs with the while loop
- counters keep track of the number of iterations and avoid infinite loops
- indentation and the colon : symbol are required for the code to run

In the next lesson, you'll put your iteration skills into practice to solve real problems.

More on Iteration

Great progress! Iteration is now in your toolbox. Let's put it into practice to solve some real-life examples.

What will this program send to the screen?

```
for i in range(3):
  print(i < 1)
```

A loop can repeat multiple statements. They all need to be indented.

How many words will this code output?

```
for i in range(3):
  print("First")
  print("Second")
```

More on Iteration

After the loop is finished, the computer will continue to execute statements in the sequence.

```
for i in range(3):
  print("A")
print("B")
```

The "Goodbye" message will only be displayed once. Why?

```
for i in range(5):
  print("Hello")
print("Goodbye")
```

More on Iteration

You've already used the comparison operators > and <. Run this code to explore some more:

```
print (5 == 5)

print (5 == 7)

print (5 != 7)

print (5 != 5)

print (5 <= 5)

print (5 >= 5)
```

Let’s look at a real-life example.

A program repeatedly asks the user to enter a password until the password is correct.

Which loop should be used?

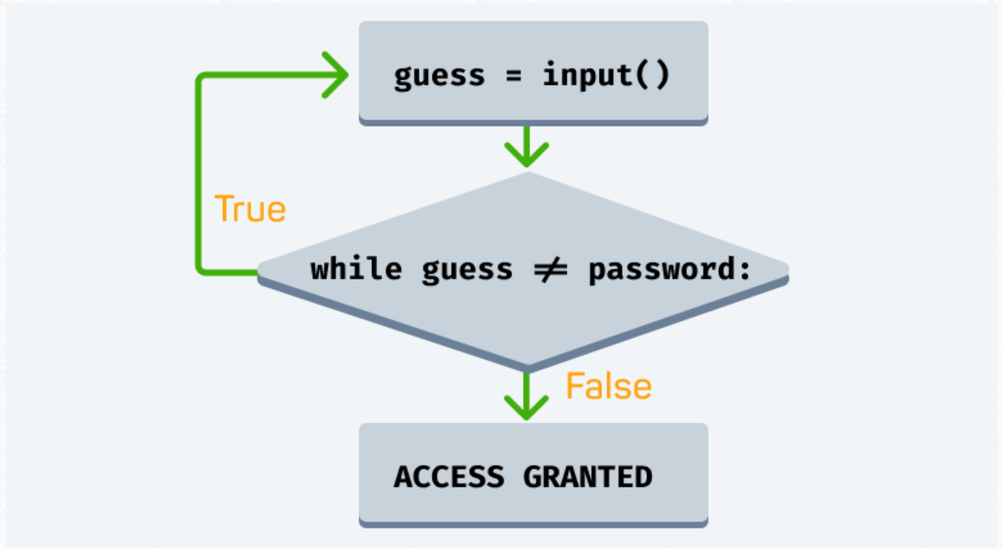
Which command would you use to allow the user to enter a password?

The user’s input is compared with the correct password. When the user correctly inputs the password, they successfully log in.

More on Iteration

What will the code output?

```
password = "SecretWord"

guess = "1234"

print(guess != password)
```



More on Iteration

When will the loop end?

```
password="SecretWord"
guess = input()
while guess != password:
  guess = input()
print("Access Granted")
```

A robot prints a label "Package A" for each of the 50 packages in a collection.

Which loop should be used to program this robot?

The robot must display "Task Complete" once all labels have been printed.

What is the error in the code?

```
for box in range(50):
  print("Package A")
  print("Task complete")
```

More on Iteration

Lesson Takeaways

Great progress! You learned that:

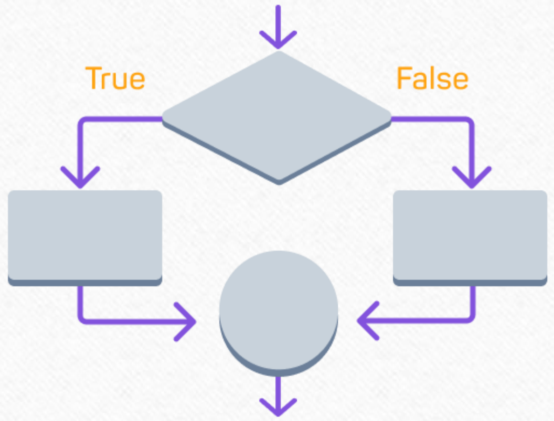
- for loops are used when the number of iterations is known
- you can solve real problems by combining comparison operations and iterations

In the next lesson, you’ll learn to use selection in your programs so machines can make decisions for you.

Conditional Statements

Selection is like a fork in the road. If allows your programs to decide which path to take.

In this lesson, you'll learn to build code that uses selection to make decisions.



Conditional Statements

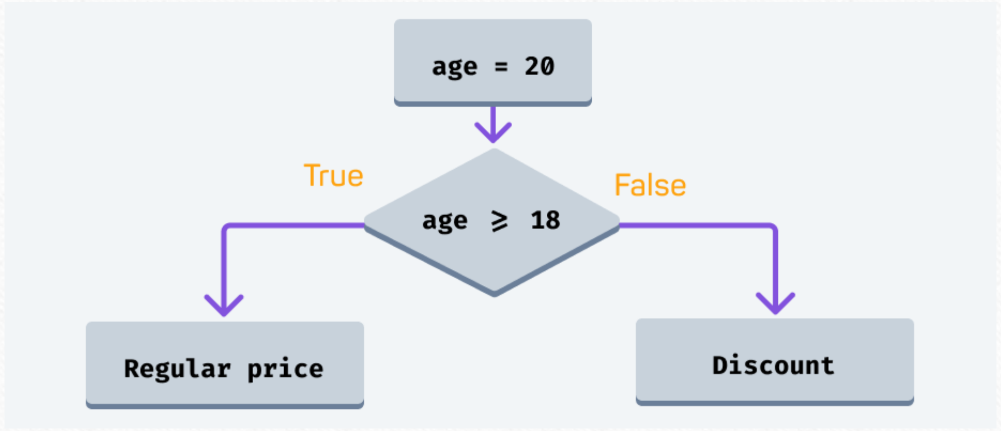
Conditional statements, or if-else statements, allow programs to perform different actions based on the conditions.

```
age = 22
if age >= 18:
  print("Regular price")
else:
  print("Discount")
```

How does the output change if you set the age to 14?

The code decides if a discount should be applied based on age.

What's the output when age is 20?



Conditional Statements

What is the output when age is 16?

```
age = 16
if age >= 18:
  print("Regular price")
else:
  print("Discount")
```

The if conditional statements starts with the keyword if followed by the condition and a colon : symbol

The else conditional statement starts with the keyword else followed by a colon : symbol

The code that gets executed within the if and else blocks must be indented. Otherwise, there will be errors.

Run the code to get the error message. Then fix the bugs

```
if age >= 18:
print("Regular price")
else:
print("Discount")
```

.

Conditional Statements

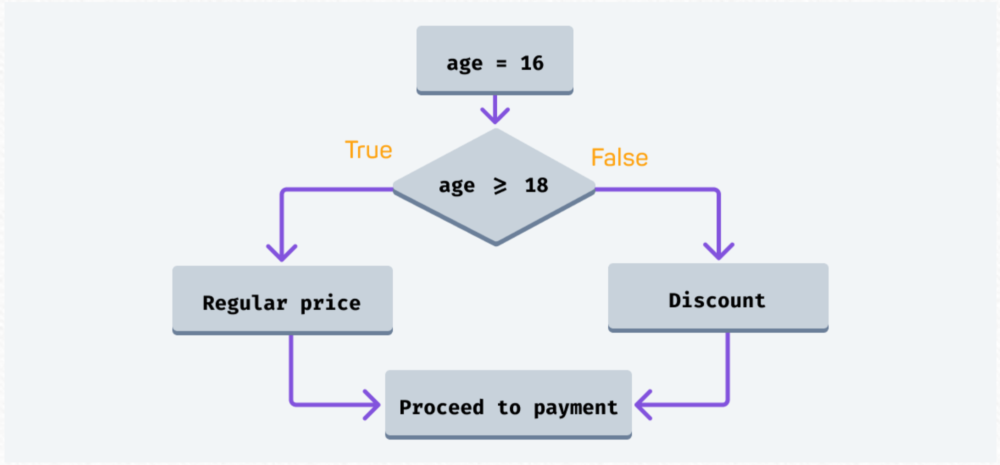
After the computer has finished executing the if-else statement, it will continue to execute any following statements in sequence

```
age = 30
if age >= 18:
  print("Regular price")
else:
  print("Discount")
print("Proceed to payment")
```

Conditional Statements

What will the code output?

```
age = 30
if  age >= 18:
  print("Regular price")
else:
  print("Discount")
print("Proceed to payment")
```



Conditional Statements

The correct use of comparison operations allows you to build programs that solve real-life problems.

Let's practice your comparison operations skills

You can build programs that make more complex decisions if you combine logical and comparison operations

is_student = True

age = 20

is_student or (age < 18)

What will this code output?

is_student = False

age = 16

print(is_student and (age < 18))

The program applies a discount if the client is under 18 or a student.

Conditional Statements

Run the code to see the output

```
if age < 18 or is_student:
  print("Discount")
else:
  print("Regular price")
```

What will the code output?

```
age = 32
is_student = True
if age < 18 or is_student:
  print("Discount")
else:
  print("Regular price")
```

Conditional Statements

Lesson Takeaways

Well done! You learned that:

- if-else statements are used to implement selection into your programs
- the colon : symbol and the use of indentation are needed to prevent errors

In the next lesson, you’ll dive deeper into selection.

More on Conditional Statements

Conditional statements allow you to program machines that make decisions.

In this lesson, you’ll learn to program more complex scenarios.

More on Conditional Statements

What will the output be?

```
temperature = 36
if temperature > 39:
  print("High temperature")
else:
  print("No fever")
```

There will be situations where you don’t need the else statement.

This program only applies a discount when the age is under 18.

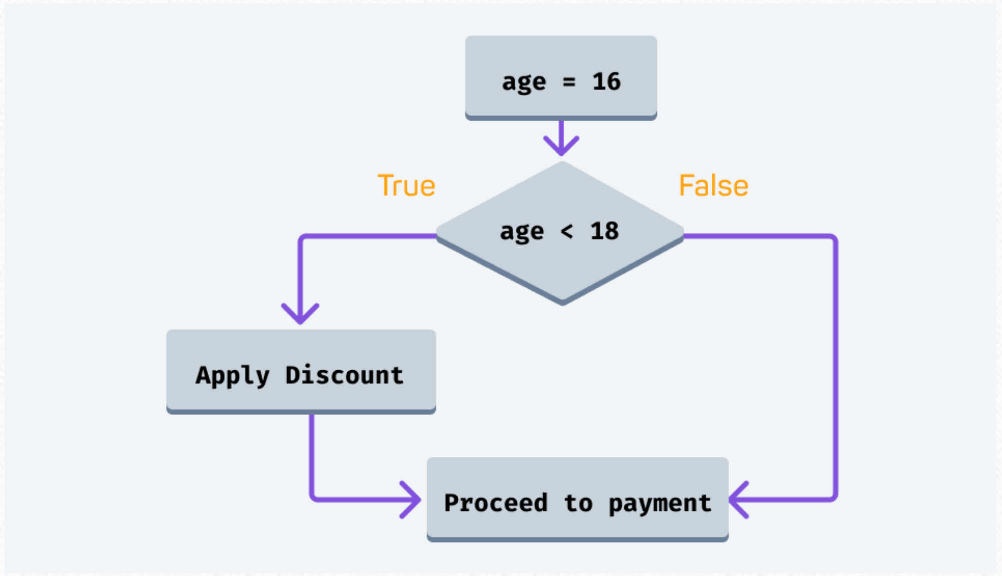
The program does nothing else if the condition is not met, so the else statement can be skipped.

```
if age < 18:
  print("Apply Discount")
print("Proceed to payment")
```

More on Conditional Statements

When the else statement is not needed, you can simplify and have the code for the selection block in 1 line.

```
age = 16

if age < 18: print("Apply Discount")

print("Proceed to Payment")
```

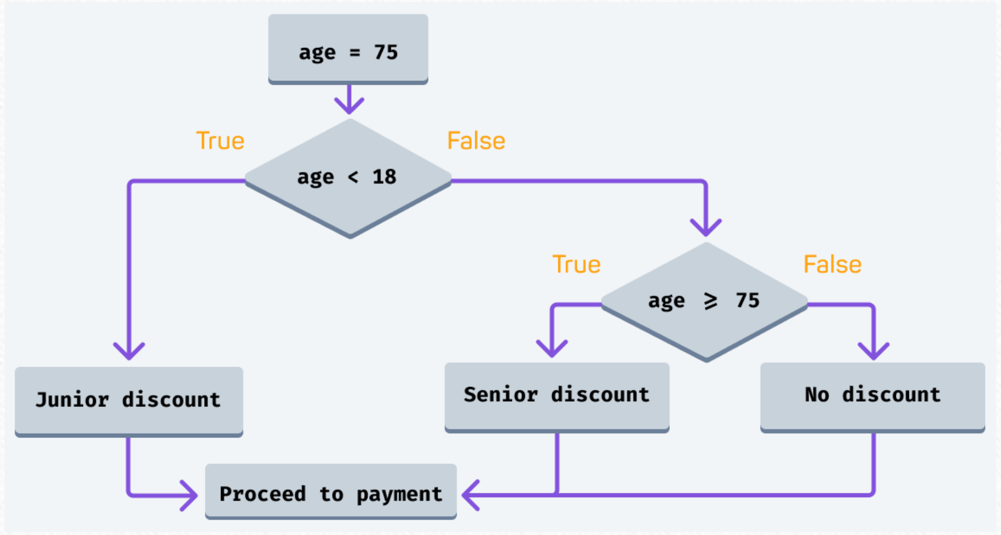


More on Conditional Statements

You can use the elif statement (short for "else if") to check for more conditions if the first condition is not met.

```
if age < 18:
  print("Junior discount")
elif age >= 75:
  print("Senior discount")
else:
  print("No discount")
```

When is the senior discount applied?



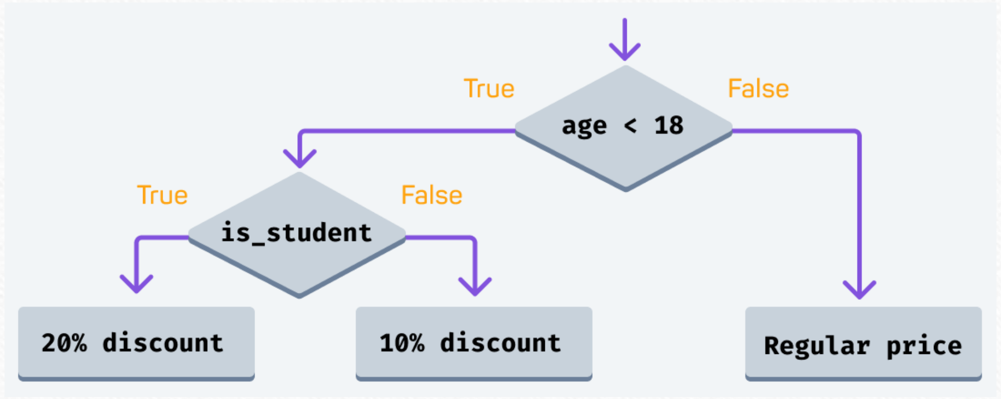
More on Conditional Statements

As with any other conditional statement, elif requires the colon : symbol and for the code that gets executed under it to be indented.

You can nest if-else statements within each other.

```
if age < 18:
  if is_student:
    print("20% discount")
  else:
    print("10% discount")
else:
  print("Regular price")
```

What would be the discount when the client is 17 and not a student?



More on Conditional Statements

Different levels of indentation are used to nested blocks

```
if age < 18:
  if is_student:
    print("20% discount")
  else:
    print("10% discount")
else:
  print("Regular price")
```

Indent the code correctly so the program runs without errors

```
if age < 18:
if is_student:
print("20% discount")
else:
print("10% discount")
else:
print("Regular price")
```

More on Conditional Statements

The program will return an error. Why?

```
if age < 18:
  print("Junior discount")
else:
  print("No discount")
```

What will the code output?

```
age = 80
if age < 18:
  print("Junior discount")
elif age >= 75:
  print("Senior discount")
print("Proceed to payment")
```

More on Conditional Statements

What will the code output?

```
age = 29
if age < 18:
  if is_student:
    print("20% discount")
else:
  print("Regular price")
print("Proceed to payment")
```

More on Conditional Statements

Lesson Takeaways

Great job! You learned that you can:

- skip the else statement when it is not needed
- check for more conditions with the elif statement
- nest selection blocks within each other

In the next lesson, you'll learn to work with collections of values in Python.

Lists

What do a shopping list, a playlist, a box of chocolates, and a bookshelf have in common? They are all collections of items.

In this lesson you will learn how to work with collections in Python.

Lists

Lists allow you to store a collection of multiple values in a single variable.

Add square brackets [ ] around the values to create a list

A list can store multiple values.

How many values are stored in the list?

```
cart = ["milk", "eggs", "apples"]
```

The different items in a list need to be separated by commas.

Lists

Just like variables, lists have a name.

What's the name of the list?

```
games = ["Snake", "Puzzle", "Chess"]
```

Lists can store any data type.

What type of values does this list store?

```
prices = [0.59, 2.99, 14.5]
```

Lists can contain values of different data types.

What data types are there in the given list?

```
movie_info = ["Inception", 2010]
```

Lists

You can place list items on separate lines to make the code easier to read

You can display a list on the screen by using the print() instruction.

```
shopping_cart = [
  "laptop",
  "smartphone",
  "headphones",
  "backpack"
]
print(shopping_cart)
```

The different items in a list need to be separated by commas.

Lists

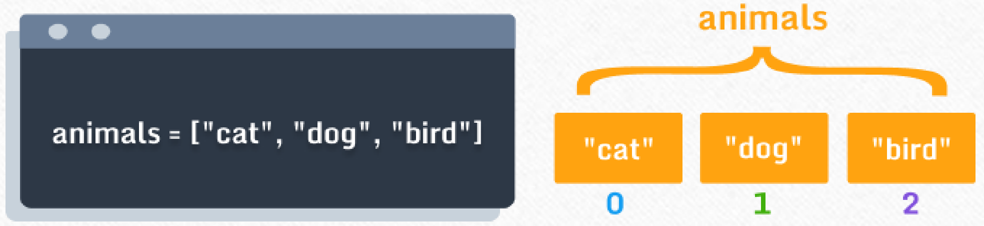
Something is wrong with the list below. Can you fix the errors?

countdown = [3 2 1]

print(countdown)

Lists are ordered collections of items. You can access items in a list using its position or index number.

Use the image to find the index for the "cat" item



```
animals = ["cat", "dog", "bird"]
```

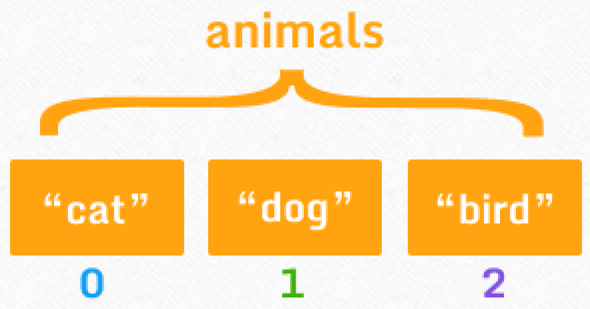
Lists

The first item in a list has an index of 0

animals = ["cat", "dog", "bird"]

A list is a sequence of values. Each value in a list has an index.

Index numbers in Python start at 0.



Lists

Lesson Takeaways

Amazing! Lists are in your coding arsenal. You learned that:

- You can store multiple values in a single variable with lists
- Lists can contain values of different data types
- Values in a list are ordered by position or index number

In the next lesson, you'll learn about indexing to perform operations with lists.

Indexing

Lists are an essential concept in programming.

They let you store multiple values in a single variable, making the program more efficient and organized.

In this lesson you'll learn to work with data that has been stored in lists.

Indexing

You can refer to the values in a list by its position (or index).

Include the index in square brackets to refer to the second value in the list

```
animals = ["cat", "dog", "bird"]
```

You can use print() to output individual values on the screen.

```
palindromes = ["radar", "level", "noon"]
print(palindromes[0])
print(palindromes[1])
```

Indexing

What will this code output?

```
animals = ["cat", "dog", "bird"]
print(animals[2])
```

What's the index of the 3rd element in the list?

```
songs = ['Trip', 'Perfect', 'Yesterday']
```

Indexing

Lists are mutable data structures. This means you can change the values in a list after it has been created.

```
nums = [8, 6, 19]

nums[0] = 1

print(nums)
```

What will this code output?

```
products = ["apples", "oranges", "bananas"]

products[2] =  "lime"

print(products[2])
```

Indexing

Indexing in Python starts at 0. What's the output of this code?

```
devices = ["TV", "Notebook", "PC"]
print(devices[1])
```

What will this code display on the screen?

```
words = ["rise", "sun", "glasses"]
print(words[1] + words[0])
```

What will this code output?

```
products = ["apples", "oranges", "bananas"]
products[1] =  "lime"
print(products[2])
```

Indexing

Lesson Takeaways

Amazing! Indexing is in your toolbox. You learned that:

- You can access the values stored in a list with indexing
- Lists are mutable, which means that they can be modified after they have been created

In the next lesson, you’ll put your indexing skills into practice.

Using Indexing

Indexing is a powerful technique that allows you to call a portion of a list.

In this lesson, you'll use indexing to work with items inside of a collection individually.

Using Indexing

Complete the code to replace "Puzzle" with "Race"

```
games = ["Snake", "Puzzle", "Shooter"]
```

You can create a list with values that have been stored in variables.

```
name = "Sarah"

age = 34

country = "Germany"

info = [name, age, country]
```

Complete the code to output the user’s name

```
name = "Lee"

country = "France"

user_info = [name, country]
```

Using Indexing

The code below is part of the program for a vending machine.

What will the code output?

```
products = ["juice", "chocolate", "water"]

user_choice = 1

print(products[user_choice])
```

Complete the code for the vending machine to take an input from the user and output the selected product

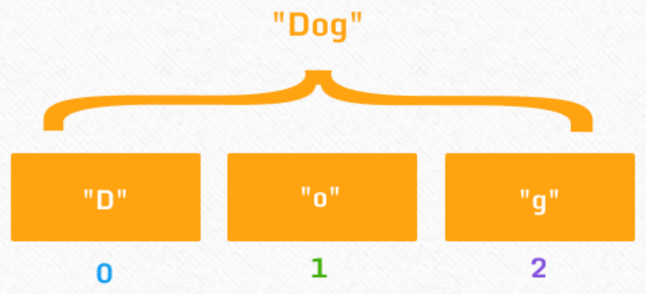
```
products = ["juice", "chocolate", "water"]
```

Using Indexing

Indexing also works with strings. You can use indexing to access individual characters in a string

animal = "Dog"

print(animal[0])



Using Indexing

A string is a sequence of characters. What character will this code output?

```
animal = "Dog"
print(animal[0])
```

The characters in a string can include spaces and punctuation marks. What character will this code output?

```
notification = "New message!"
print(notification[4])
```

Using Indexing

What character will the code output?

```
characters = "!#- ?"
print(characters[4])
```

Strings are immutable, which means that you can’t change the characters in a string. If you try to change a string you’ll get an error.

```
word = "car"
word[2] = "t"
```

True or false? You can change the values stored in a list after it has been created

Using Indexing

What will this code display on the screen?

```
x = "arctic"

print(x[2] + x[0] + x[3])
```

The code will result in an error. Why?

```
month = "august"

month[0] = "A"
```

Using Indexing

Lesson Takeaways

Amazing! You learned that:

- You can create a list with values that have been stored in variables
- You can use indexing with strings
- Strings are immutable

In the next lesson, you'll learn to extract a section of the data stored in a sequence.

Slicing

Indexing allows you to access individual values from a sequence.

In this lesson you'll learn to extract, modify and replace a specific range of elements from sequences with a new technique: slicing.

Slicing

Both lists and strings are Python sequences. Which means that their content is ordered.

True or False? You can use indexing with both lists and strings

What will this code output?

```
movies = ["Avengers", "Avatar", "Titanic"]
print(movies[1])
```

The following code represents greeting options in a chatbot program.

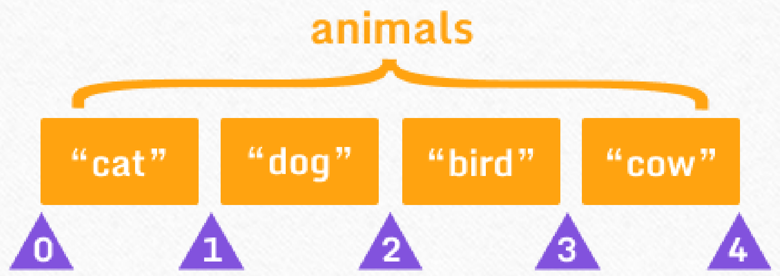
```
greet = ["Hello", "Hey", "Good day"]
```

Complete the code to output Good day

Slicing

Slicing allows you to extract a portion of a list. Starting and stopping indices are separated by a colon :

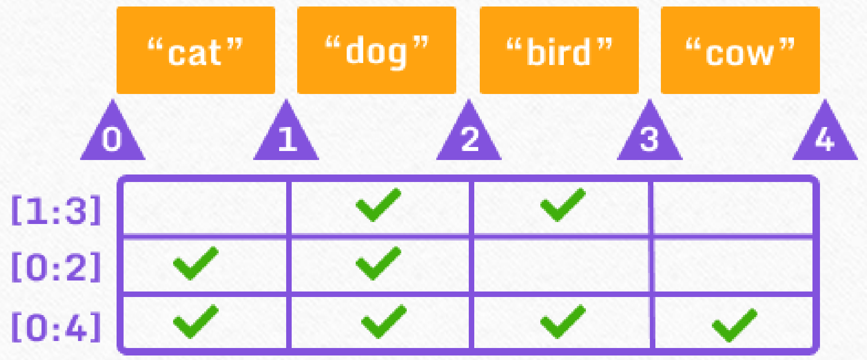
animals[1:3]



Slicing

The starting index is inclusive. The stopping index is exclusive.

animals[0:2]



Slicing

What will this code output?

```
animals =['dog', 'cat', 'bird', 'cow']
print(animals[2:4])
```

What will the code output?

```
animals =['dog', 'cat', 'bird', 'cow']
print(animals[0])
```

Slicing

Slicing also works with strings.

Slicing a list produces another list

colors = ['red', 'green', 'blue', 'yellow']

Complete the code to output a list that contains 'green' and 'blue'

What will this code output?

colors = ['red', 'green', 'blue', 'yellow']

print(colors[2:3])

Slicing

Slicing a string produces another string

color = 'orange'

Complete the code to output or

What will this code output?

color = 'pink'

print(color[1:4])

How many values will be extracted from the list

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter']

print(planets[2:4])

Slicing

Lesson Takeaways

Great job! You learned that:

- You can extract a section of a sequence with slicing
- Slicing a list produces another list
- Slicing a string produces another string

In the next lesson, you'll put your slicing skills into practice.

Using Slicing

Slicing is a powerful technique that allows you to precisely select data from a sequence.

In this lesson, you'll learn some new slicing techniques.

Using Slicing

Complete the code to output the 2nd and 3rd items from the shopping cart list

```
cart = ['lamp', 'candles', 'chair', 
'carpet']
```

When slicing, you can omit the starting index. This means that you'll be slicing from the very first element.

```
cart = ['lamp', 'candles', 'chair', 
'carpet']

print(cart[:3])
```

Using Slicing

What will this code output?

```
cart = ['lamp', 'candles', 'chair',
'carpet']

print(cart[:2])
```

Let's apply the technique to a string.

What will this code output?

```
vehicle = 'motorbike'

print(vehicle[:5])
```

Using Slicing

When slicing, you can omit the stopping index. This means that you'll be slicing until the very last element.

```
cart = ['lamp', 'candles', 'chair', 'carpet']

print(cart[1:])
```

What will this code output?

```
cart = ['lamp', 'candles', 'chair', 'carpet']

print(cart[2:])
```

Using Slicing

What will this code output?

```
vehicle = 'motorbike'

print(vehicle[5:])
```

What will this code output?

```
vehicle = 'motorbike'

print(vehicle[:])
```

Using Slicing

What will this code output?

```
platform = ['iOS', 'Android', 'Web']

print(platform[:])
```

This code will output a list. How many values will it contain?

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars']

print(planets[2:])
```

Using Slicing

What will this code output?

```
char = ['A', 'B', 'C', 'D', 'E']
print(char[2:4])
```

This code will output 3 ordered values in a list

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars']
print(planets[:3])
```

What elements will be included in the output?

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars']
print(planets[2:])
```

Using Slicing

Lesson Takeaways

You did it!. This is what you learned:

- You can omit the starting index when slicing a list or a string
- You can omit the stopping index when slicing a list of a string

In the next lesson, you'll take your slicing skills to the next level and will learn some advanced techniques.

Advanced Slicing and Indexing

Indexing and slicing are powerful techniques that allow you to fully control the data that is extracted from a sequence.

In this lesson, you’ll take your indexing and slicing skills to the next level.

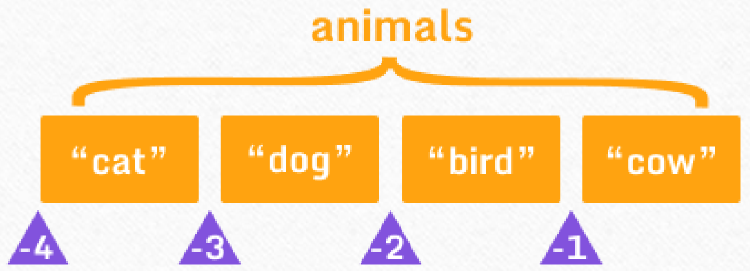
Advanced Slicing and Indexing

You can use indexing and slicing because lists and strings are ordered sequences.

Python supports "indexing from the end", that is, negative indexing. This means the last value of a sequence has an index of -1.

```
animals =["dog" , "cat", "bird", "cow"]
```

```
print(animals[-1])
```



Advanced Slicing and Indexing

What will this code output?

```
c = ['$', '£', '€', '¥']
print(c[-1])
```

You can combine positive with negative indices when slicing.

What will this code output?

```
c = ['$', '£', '€', '¥']
print(c[1:-1])
```

What will this code output?

```
c = ['$', '£', '€', '¥']
print(c[0:-2])
```

Remember lists are mutable. You can change the values in a list after it has been created.

What will the code output?

```
c = ['$', '£', '€', '¥']
c[1] = '₣'
print(c)
```

Advanced Slicing and Indexing

What will the code output?

```
c = ['$', '£', '€', '¥']
c[:2] = ['Fr', '฿']
print(c)
```

What will the code output?

```
vehicle = 'airplane'
vehicle[:3] = 'water'
```

Why will this code result in an error?

```
vehicle = 'airplane'
vehicle[:3] = 'water'
```

Advanced Slicing and Indexing

Lesson Takeaways

You did it! You learned that:

- Python supports “indexing from the end”, that is, negative indexing
- The last value of a sequence has an index of -1

In the next lesson, you’ll learn how to make your code more efficient with functions.

Functions

In this lesson, you will learn about functions.

Functions can be thought of as reusable blocks of code that perform specific tasks.

They help break our program into smaller and modular chunks, avoid repetition and make the code reusable.

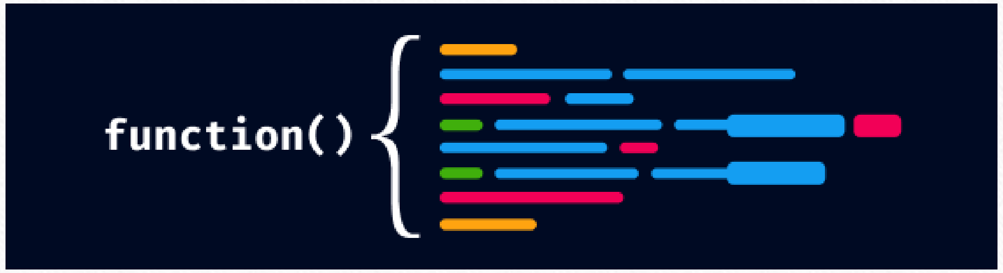
You have already seen (and used) functions in previous lessons. print(), input(), and type() are examples of functions.

Functions

A function performs a task.

A function contains the code required to perform a task. To execute this code, the function needs to be called.

Every function has a name. The code contained within a function is only executed when the function is called.



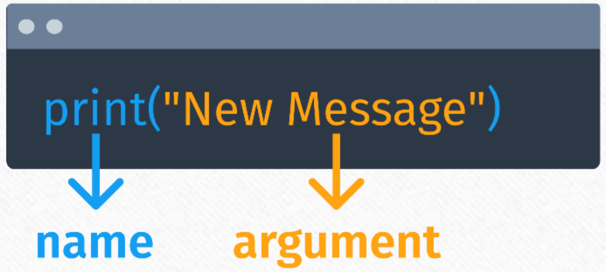
Functions

As an example, the print() function contains hundreds of lines of code with all the instructions required to take a message, process it and display it on the screen.

Functions make the code reusable. The same function can be called multiple times.

Functions require information to be passed in order for the task to be completed. We pass information into functions as arguments.

Arguments are put inside a set of parentheses () following the function name.



```
print("New Message")
```

Functions

Another example of a function that you have used is range().

The range() function generate a sequence of numbers.

The range() function takes in a number as an argument

range(3)

Identify the elements of the function

range(5)

Functions rely on their arguments to work. Missing arguments can result in errors.

range()

This code will return …

Functions

A function can take multiple arguments. We can pass multiple arguments to the print() function to display multiple values on the screen.

print("Your seat:", 4)

Displayed values are separated with a whitespace.

Select the number of arguments that are being passed to the function

print("Credit:", 98)

The print function can take arguments from different data types.

Identify the data types of the arguments

print("Score:", 35)

Multiple arguments in a function are separated with a comma ,

print() and range() are examples of built-in functions. Python has many built-in functions which you can re-use in your code by calling them.

Functions

Lesson Takeaways

Great job! You learned that:

- Functions are reusable blocks of code that perform specific tasks
- Functions make your code organized and reusable
- Information to complete the task is passed as arguments

In the next lesson, you'll dive deeper into built-in functions to become a more effective coder.

Function Arguments

Built-in functions like print() and range() are blocks of code created by others to solve specific problems.

This simplifies the code a lot as there is no need for you to create code from scratch each time, all you need to do is to call them!

In this lesson, you'll delve deeper into the built-in functions and explore how they can enhance your coding efficiency and productivity.

Function Arguments

How many arguments does this function have?

```
print("Name:", "Anna", "Age:", 25)
```

The print() function can take arguments with any data type:

```
print("Online:", True) # string and boolean
print("Credit:", 385.91) # string and float
print(2, " bananas") # integer and string
```

Identify the data types of the arguments

```
print("Weight:" , 77.6)
```

Function Arguments

Functions can take different types of operations as arguments.

As an example, the print() function can accept arithmetic, logical and comparison operations.

```
print(55 * 3) #arithmetic
print(5 > 7) #comparison
print(True and False) #logical
print("motor" + "bike") #concatenation
```

What will this code output?

```
x = "air"
y = "plane"
print(x + y)
```

Different functions accept different types of arguments. For example, the range() function accepts only integers.

Function Arguments

You can use values stored in variables as arguments.

What will this code output?

```
balance = 304
print("Money in account:", balance)
```

What will this code display on the screen?

```
name = "Tom"
country = "France"
age = 35
print(country)
```

Function Arguments

Complete the code to display Table 5 on the screen

```
item = "Table"

num = 5
```

The arguments passed to a function must be separated by a comma, otherwise it'll cause an error.

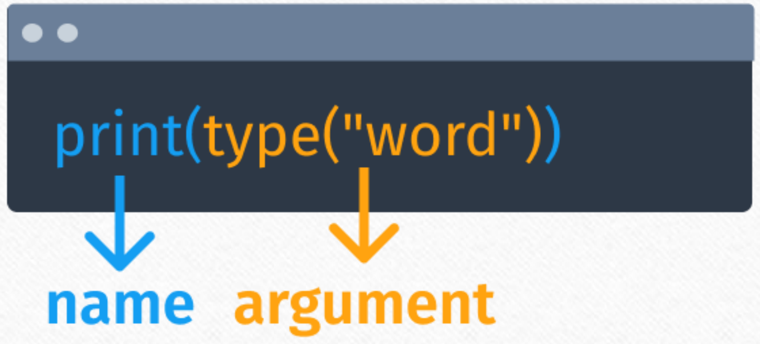
What will this code generate?

```
name = "Anna"

surname = "Anderson"

print(name surname)
```

Function Arguments

A function can be the argument for another function.

print(type("word"))


```
print(type("word"))
```

Function Arguments

int(), str(), and float() instructions are functions used for data conversion.

Your code will result in errors if you pass the incorrect data type as an argument. Some functions require specific data types as arguments.

For example, the int() function won't be able to convert non-numeric characters into numbers and it'll result in an error.

```
int('pencil') # error
```

What will this code output?

```
num = '45' # string
print(int(num) + 3)
```

Function Arguments

Lesson Takeaways

Awesome! You have learned that:

- You can use variables as arguments for functions
- A function can be an argument for another function
- Different functions require specific data types as arguments

In the next lesson, you'll learn to work more effectively with text data by using string functions.

String Functions

Correctly handling text data is a key skill all coders need to master. String functions will enhance your productivity when working with text.

In this lesson, you'll learn to use built-in string functions.

String Functions

A string is a sequence of characters in quotes.

Strings functions make it easier to work with strings.

The functions lower() and upper() allow the case of a string to be altered quickly,

so that all the characters are lower- or uppercase, respectively.

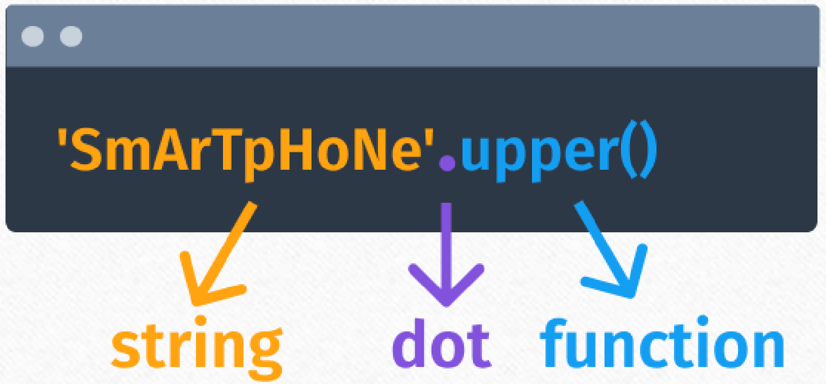
String Functions

Give it a try

'SmArTpHoNe'.lower()

'SmArTpHoNe'.upper()

upper() and lower() functions can only be used on strings. Calling a function using the dot . notation makes this explicit.



```
'SmArTpHoNe'.upper()
```

String Functions

Many built-in functions need to be called using the dot notation. The dot notation shows belonging.

Remember that the print() function is needed to output results on the screen.

Remember you can use variables in combination with functions.

The capitalize() function will save you time when you need to convert the first character of a string to uppercase, while making the remaining characters lowercase.

```
"happy birthday".capitalize()
```

String Functions

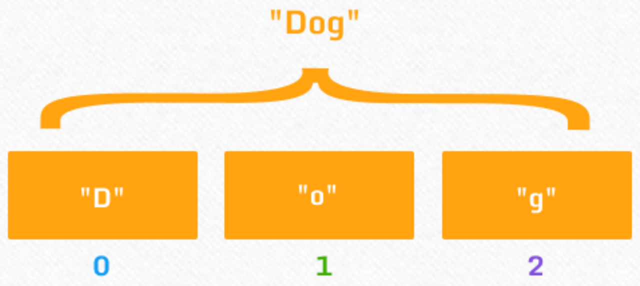
Strings are immutable. This means that once created, you can’t change them.

Strings are immutable and functions won't change them. You'll need to store the modified string in a variable to keep it.

item = "smartwatch"

item2 = item.upper()

What's the index of the letter g in the string Dog?



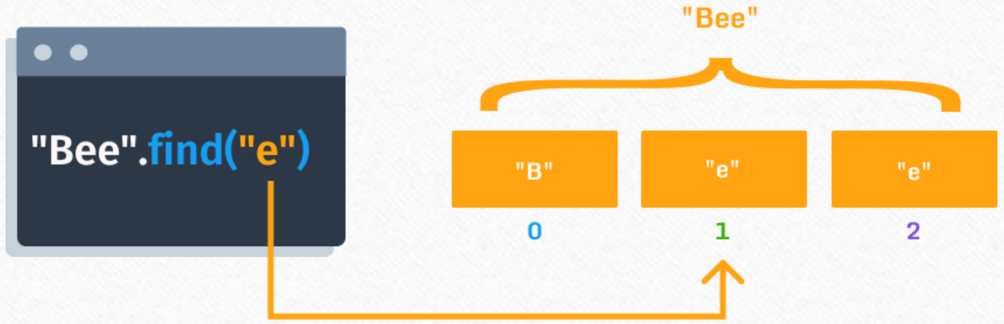
String Functions

The find() function checks if a character (or a pattern of characters) is present in a string.

The function returns the index (position) of the given value.

If the given value is present multiple times, the function will return the lowest index (first occurrence).

"Bee".find("e")


```
"Bee".find("e")
```

String Functions

The find() function acts on a string so it's called using the dot notation.

What will this function return?

"robot".find("o")

Remember that the print() function is needed to output information on the screen.

What will this code output?

print("robot".find("t"))

find() is an example of a function that will return an error if you don't include an argument between the brackets.

Without a value to find in the string the function won't be able to complete its task.

String Functions

This won't happen with upper(), lower() and capitalize(). No additional information is needed for the function to complete its task.

'roBot'.find() # error

'roBot'.upper() # no error

find() will turn -1 if the value can't be found in the string.

What will this code return?

print("robot".find("A"))

What will this code output?

word = 'vehicle'

print(word.find('r'))

What will this code output?

word = 'vehicle'

print(word.find())

String Functions

Lesson Takeaways

Great job! You learned that:

- upper(), lower(), and capitalize() functions are used to change the case of a string
- The find() function searches for a value in a string returns the index of the first occurrence

In the next lesson, you'll learn to use functions that act on lists.

List Functions

Lists are an essential concept in coding.

They let you store multiple values in a single variable, making the program more efficient and organized.

In this lesson you'll learn to use list built-in functions.

List Functions

Lists are ordered sequence of items.

What's the index for the value bicycle?

items = ["bike", "car", "bicycle", "scooter"]

Use indexing to complete the code to display Avengers on the screen

movies = ["Avatar", "Titanic", "Avengers"]

List Functions

len() is one of the most useful built-in functions. It can be applied to lists. It stands for length

and returns the number of items in a collection.

```
movies = ["Avatar", "Titanic", "Avengers"]
len(movies)
```

The len() function is not specific to lists. It accepts as an argument any collection of items. This includes strings.

What will the code output?

```
movie = "Avatar"
print(len(movie))
```

len() is a very versatile function that accepts a wide range of arguments.

It's not specific to a data type or particular object and that's why you should not use dot notation to call this function.

List Functions

The append() function acts on a list to add a new item to the end.

append() is called using the dot notation because it's specific to lists.

songs = ["Yesterday", "Hello", "Believer"]

songs.append("Imagine")

append() modifies a list. This means that the new lengthened list will be stored with the same name. This is possible because lists are mutable.

The append() function acts on a list. If you try to apply append() on a string you'll get an error.

This happens because are immutable.

List Functions

The append() function is specific to lists. You must use dot notation to call append() on a list.

The append() function adds an item to the end of a list.

What will this code output?

```
movies = ["Avatar", "Titanic", "Avengers"]
movies.append("Alien")
print(movies[3])
```

Complete the code to add "Thriller" to the end of the list

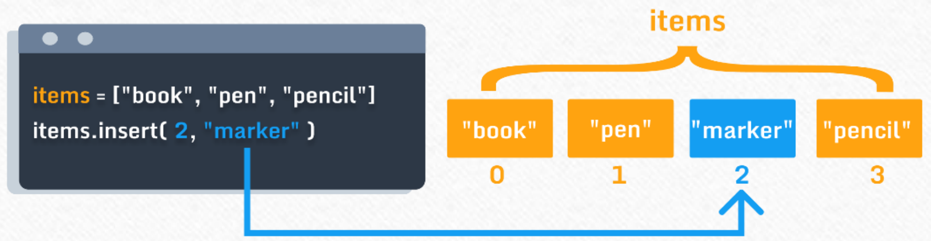
```
genres = ["Sci-Fi", "Fantasy", "Drama"]
```

List Functions

The insert() function allows you to add an element to a list, at a specific position.

items = ["book", "pen", "pencil"]

items.insert(2,"marker")



```
items = ["book", "pen", "pencil"]
items.insert( 2, "marker" )
```

List Functions

insert() takes 2 arguments. The first is the index (where to insert) and the second is the item (what to insert).

Complete the code to insert Green as the first item in the list

```
colors = ["Red", "Blue", "Yellow"]
```

Identify the elements with their indices after executing the code

```
colors = ["Red", "Blue", "Yellow"]
colors.insert(2, "Green");
```

What will this code display on the screen?

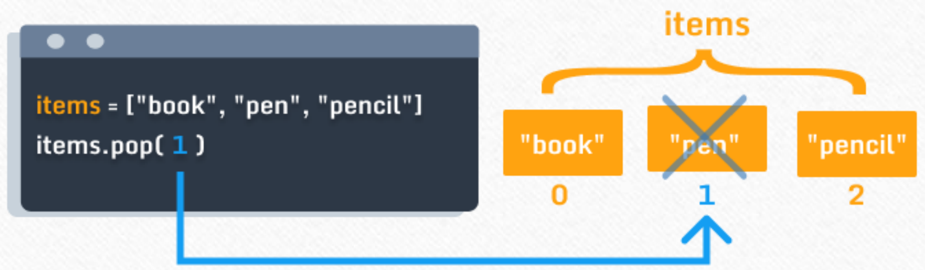
```
colors = ["Red", "Blue", "Yellow"]
colors.insert(2, "Green")
colors.append("Black")
print(colors[3])
```

List Functions

The pop() function removes an element from a list.

That position indicated by the index is the only argument that the pop() function accepts.

items.pop(1)



```
items = ["book", "pen", "pencil"]
items.pop( 1 )
```

List Functions

What will this code output?

```
class = ["Maths","English","Physics"]
class.pop(2)
print(class)
```

Complete the code to remove the third element and then add 95 to the end of the list.

```
points = [15, 36, 74, 88]
```

Complete the code to remove the "Toyota" element from the list. Then insert "Tesla" as the new third element, and display the updated list.

```
cars = ["BMW", "Toyota", "Audi",
"Honda"]
```

List Functions

Lesson Takeaways

Great job! You learned that:

- len() returns the number of items in a sequence
- append() adds an item to the end of a list
- insert() and pop() bring in and take out list items at specific positions

In the next lesson, you'll learn to create your own functions to perform specific tasks.

Custom Functions

Built-in functions can save you a lot of work.

There will be situations where you'll need to create your own custom functions to solve specific tasks.

In this lesson, you'll learn to create your own functions to make your code efficient and reusable.

Custom Functions

A function is a reusable block of code.

To use your own functions, you need to define them first.

Once a function has been defined, you can call it as many times as you need.

The greet() function contains the code to display a nice message when called

```
def greet():
  print("Hello from a function")
  print("Have a great day")
```

Once a function has been defined, you can call it as many times as you need to reduce the length of your code.

Custom Functions

Complete the code to call the defined function

```
def greet():
  print("Hello from a function")
  print("Have a great day")
```

The body of a function contains the reusable code that is executed when the function is called.

The code for the body of a function must be indented.

The body for the greet() function consists of 2 lines of code.

When a function is defined, you need to make sure parentheses () are added after the name.

A colon : must be added at the end of the definition line.

Indentation, parentheses and the colon are needed for the code to run without errors.

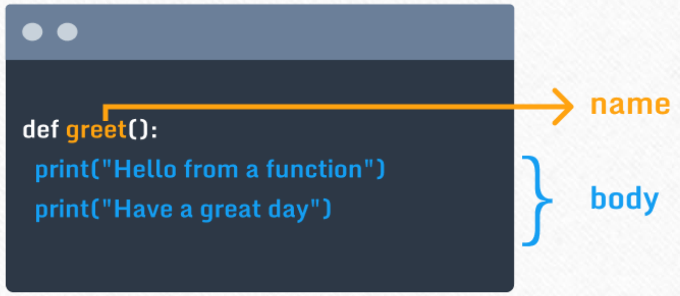
Custom Functions

Fix the bugs in this code

def greet

print("Hello from a function")

print("Have a great day")



```
def greet():
 print("Hello from a function")
 print("Have a great day")
```

Custom Functions

A function performs a task. The greet() function performs the task of displaying a nice message.

You can define functions that take any number of arguments (including zero).

Arguments are put inside the parentheses () following the function name.

```
def greet():
  print("Hello from a function")
  print("Have a great day")
```

The greet() function requires no argument.

A function might require arguments to complete its tasks. Arguments are put inside the parentheses () following the function name

```
def personal_greet(name):
  print("Hello", name)
  print("Have a great day")
```

Complete the code to call the defined function and display a personalized message for James

```
def personal_greet(name):
  print("Hello", name)
  print("Have a great day")
```

Custom Functions

Fix the code to define a function that takes a number and outputs its double on the screen

```
def double(number)

print(number*2)
```

Functions must be defined before they can be called.

Fix the code so it doesn't result in errors

```
#function call

double(6)

def double(number):

print(number*2)
```

Custom Functions

A function can take as many arguments as needed to complete the task.

The code defines a function that displays the Body Mass Index (BMI)

How many arguments does the function have?

```
def bmi(weight, height):
    index = weight / (height * height)
    print(index)
```

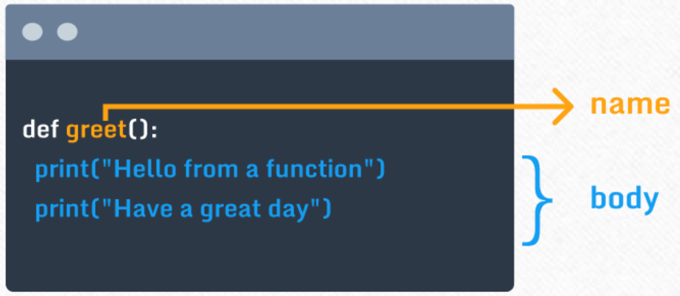
What are the arguments that the function requires to complete the task?

```
def bmi(weight, height):
    index = weight / (height * height)
    print(index)
```

Custom Functions

When calling a function, you need to use the same number of arguments that have been defined, in the same order.

Complete the code to call the function and calculate the BMI for a user who is 1.75 m tall and weighs 79 kg



```
def greet():
  print("Hello from a function")
  print("Have a great day")
```

Custom Functions

The result of a function can be sent back to the caller code with the return statement.

This is particularly helpful when you need to continue using the result value in your program.

```
def bmi(weight, height):
    index = weight / (height * height)
    return index
```

Custom Functions

Lesson Takeaways

Great job! You can create your own functions now.

You learned that:

- def defines a new function
- The function body contains the code that is executed when the function is called
- return sends values back to the caller code so they can be stored and used in the program

In the next lesson, you'll dive deeper into custom functions to make your code more efficient.

More on Custom functions

Custom (or user-defined) functions help to decompose a large program into small segments to make it easy to understand, maintain and debug.

In this lesson, you'll dive deeper into custom functions.

More on Custom functions

A function argument is a value passed when calling a function.

A function can return multiple values. The function rect() helps a real estate agency calculate the area and perimeter of a rectangular parcel of land.

It takes the two dimensions of the parcel as arguments

```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
  return area, perimeter
```

Multiple return values need to be separated by commas.

More on Custom functions

Complete the code to define the rect() function and return 2 values

```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
```

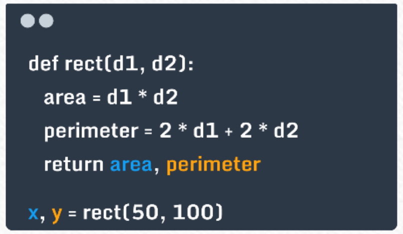
How many values will the new rect() function return when called?

```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
  price = 1000 * area
  return area, perimeter, price
```

More on Custom functions

When a function returns multiple values, they can be stored in multiple variables, with 1 line of code.

Match the variable with the return value it is storing



```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
  return area, perimeter

x, y = rect(50, 100)
```

More on Custom functions

To create multiple variables in 1 statement, separate them with commas.

Match the variable with the value it is storing

```
a, b = 7, 12
```

A function can return an error if something goes wrong in its body.

Passing argument values in the incorrect data type is a common source of errors.

Run the code to get the error. Can you fix the bug?

```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
  return area, perimeter
x, y = rect("five")
```

More on Custom functions

Valid arguments for the rect() function are…

```
def rect(d1, d2):
  area = d1 * d2
  perimeter = 2 * d1 + 2 * d2
  return area, perimeter
```

It's up to you as a coder to define what operations happen inside a function, what data types can be handled and what the function returns.

The profitable() function determines if buying a parcel of land is a good investment for a real estate agency in a particular location

```
def profitable(d1, d2):
  area = d1 * d2
  invest = area > 700
  return invest
```

More on Custom functions

Match the type of values that this functions takes in and out

```
def profitable(d1, d2):
  area = d1 * d2
  invest = area > 700
  return invest
```

The execution of the code inside a function ends when a value is returned. Any additional lines of code after the return line will be ignored

```
def rect(d1, d2):
  area = 0
  return area
  #End of function execution
  area = d1 * d2
```

More on Custom functions

What will this function return when called?

```
def rect(d1, d2):
  area = 0
  return area
  area = d1 * d2
```

Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value

```
def greet(name="Guest"):
  print("Welcome", name)
greet() # Welcome Guest
greet("John") # Welcome John
```

More on Custom functions

What will this code output when executed?

```
def greet(name="Guest"):
  print("Welcome", name)
greet()
```

The default value is used only if no other value has been passed as an argument when the function is called.

What will this code output?

```
def greet(name="Guest"):
  print("Welcome", name)
greet("Anna")
```

More on Custom functions

Lesson Takeaways

Congratulations! You completed this course.

In this lesson you learned that:

- A function can return multiple values
- Defining a function includes deciding the data types it can take in, handle and return
- Default values make arguments optional when calling a function

What course will you take next? Share your thoughts with your fellow learners.

Introduction of Collection Types

Welcome to the Intermediate Python course.

In this course, you will learn about Python collection types, lambda functions, generators, decorators, OOP, and much more.

We will also build real-world projects and solve several programming challenges.

This course is designed for intermediate Python developers who already know the basics of Python and want to expand their knowledge.

Introduction of Collection Types

Let's check your Python knowledge!

What is the output of this code?

```
n = [2, 4, 6, 8]
res = 1
for x in n[1:3]:
  res *= x
print(res)
```

Dictionaries

Python provides a number of built-in collection types, to store multiple values.

Lists are one of these collection types, and they allow you to store indexed values:

```
x = ['hi', 'hello', 'welcome']
 print(x[2])
```

Each item of a list has an index, which is automatically set.

Dictionaries

Dictionaries are another collection type and allow you to map arbitrary keys to values.

Dictionaries can be indexed in the same way as lists, using square brackets containing keys.

Example:

```
ages = {
  "Dave": 24,
  "Mary": 42,
  "John": 58
}
print(ages["Dave"])
print(ages["Mary"])
```

Dictionaries

Each element in a dictionary is represented by a key:value pair.

Only immutable objects can be used as keys to dictionaries. Immutable objects are those that can't be changed.

So far, the only mutable objects you've come across are lists and dictionaries.

```
bad_dict = {
  [1, 2, 3]: "one two three",
}
```

Since lists are mutable, the code above throws an error.

This means that you can use strings, integers, booleans, and any other immutable type as dictionary keys.

Dictionary Functions

To determine whether a key is in a dictionary, you can use in and not in, just as you can for a list.

Example:

```
nums = {
  1: "one",
  2: "two",
  3: "three",}
print(1 in nums)
print("three" in nums)
print(4 not in nums)
```

Run the code and see how it works!

Dictionary Functions

A useful dictionary function is get. It does the same thing as indexing, but if the key is not found in the dictionary it returns another specified value instead.

Example:

```
pairs = {
   1: "apple",
   "orange": [2, 3, 4],
   True: False,
   12: "True",}
print(pairs.get("orange"))
print(pairs.get(7, 42))
print(pairs.get(12345, "not found"))
```

Dictionary Functions

To determine how many items a dictionary has, use the len() function

What is the result of this code?

```
fib = {1: 1, 2: 1, 3: 2, 4: 3}
print(fib.get(4, 0) + fib.get(7, 5))
```

Task

You are working on data that represents the economic freedom rank by country.

Each country name and rank are stored in a dictionary, with the key being the country name.

Complete the program to take the country name as input and output its corresponding economic freedom rank.

In case the provided country name is not present in the data, output "Not found".

Recall the get() method of a dictionary, that allows you to specify a default value.

```
data = {
    'Singapore': 1,
    'Ireland': 6,
    'United Kingdom': 7,
    'Germany': 27,
    'Armenia': 34,
    'United States': 17,
    'Canada': 9,
    'Italy': 74}
```

Tuples

Tuples are very similar to lists, except that they are immutable (they cannot be changed).

Also, they are created using parentheses, rather than square brackets.

Example:

```
words = ("spam", "eggs", "sausages")
```

You can access the values in the tuple with their index, just as you did with lists:

You can access the values in the tuple with their index, just as you did with lists:

```
print(words[0])
```

Tuples

Trying to reassign a value in a tuple causes an error.

```
words[1] = "cheese"
```

Like lists and dictionaries, tuples can be nested within each other.

Tuples can be created without the parentheses by just separating the values with commas.

Example:

```
my_tuple = "one", "two", "three"
print(my_tuple[0])
```

What is the result of this code?

```
tuple = (1, (1, 2, 3))
print(tuple[1])
```

Task

You are given a list of contacts, where each contact is represented by a tuple, with the name and age of the contact.

Complete the program to get a string as input,

search for the name in the list of contacts and output the age of the contact in the format presented below:

Sample Input

John

Sample Output

John is 31

```
contacts = [
    ('James', 42),
    ('Amy', 24),
    ('John', 31),
    ('Amanda', 63),
    ('Bob', 18)
]
```

Tuple Unpacking

Tuple unpacking allows you to assign each item in a collection to a variable.

Example:

```
numbers = (1, 2, 3)
a, b, c = numbers
print(a)
print(b)
print(c)
```

Tuple Unpacking

This can be also used to swap variables by doing a, b = b, a , since b, a on the right hand side forms the tuple (b, a) which is then unpacked.

What is the value of y after this code runs?

```
x, y = [1, 2]

x, y = y, x
```

Tuple Unpacking

A variable that is prefaced with an asterisk (*) takes all values from the collection that are left over from the other variables.

Example:

```
a, b, *c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
print(b)
print(c)
print(d)
```

c will get assigned the values 3 to 8.

What is the output of this code?

```
a, b, c, d, *e, f, g = range(20)
print(len(e))
```

Task

Tuples can be used to output multiple values from a function.

You need to make a function called calc(),

that will take the side length of a square as its argument and return the perimeter and area using a tuple.

The perimeter is the sum of all sides, while the area is the square of the side length.

Sample Input

3

Sample Output

Perimeter: 12

Area: 9

The given code takes a number from user input, passes it to the calc() function, and uses unpacking to get the returned values.

```
def calc(x):
    #your code goes here
side = int(input())
p, a = calc(side)
print("Perimeter: " + str(p))
print("Area: " + str(a))
```

Sets

Sets are similar to lists or dictionaries.

They are created using curly braces, and are unordered, which means that they can't be indexed.

Due to the way they're stored, it's faster to check whether an item is part of a set using the in operator, rather than part of a list.

```
num_set = {1, 2, 3, 4, 5}

print(3 in num_set)
```

Sets

Sets cannot contain duplicate elements.

What is the output of this code?

```
letters = {"a", "b", "c", "d"}
if "e" not in letters:
  print(1)
else:
  print(2)
```

Sets

You can use the add() function to add new items to the set, and remove() to delete a specific element:

```
nums = {1, 2, 1, 3, 1, 4, 5, 6}

print(nums)

nums.add(-7)

nums.remove(3)

print(nums)
```

Sets

Duplicate elements will automatically get removed from the set.

The len() function can be used to return the number of elements of a set.

Sets can be combined using mathematical operations.

The union operator | combines two sets to form a new one containing items in either.

The intersection operator & gets items only in both.

The difference operator - gets items in the first set but not in the second.

The symmetric difference operator ^ gets items in either set, but not both.

Sets

```
first = {1, 2, 3, 4, 5, 6}
second = {4, 5, 6, 7, 8, 9}
print(first | second)
print(first & second)
print(first - second)
print(second - first)
print(first ^ second)
```

Tap Try It Yourself to play around with the code!

What is the output of this code?

```
a = {1, 2, 3}
b = {0, 3, 4, 5}
print(a & b)
```

Task

You are working on a recruitment platform, which should match the available jobs and the candidates based on their skills.

The skills required for the job, and the candidate's skills are stored in sets.

Complete the program to output the matched skill.

You can use the intersect operator to get the values present in both sets.

```
skills = {'Python', 'HTML', 'SQL', 'C++', 'Java', 'Scala'}

job_skills = {'HTML', 'CSS', 'JS', 'C#', 'NodeJS'}
```

List Comprehensions

List comprehensions are a useful way of quickly creating lists whose contents obey a rule.

For example, we can do the following:

```
# a list comprehension
cubes = [i**3 for i in range(5)]
print(cubes)
```

List comprehensions are inspired by set-builder notation in mathematics.

What does this list comprehension create?

```
nums = [i*2 for i in range(10)]
```

List Comprehensions

A list comprehension can also contain an if statement to enforce a condition on values in the list.

Example:

```
evens=[i**2 for i in range(10) if i**2 % 2 == 0]

print(evens)
```

Run the code and see how it works!

Task

Given a word as input, output a list, containing only the letters of the word that are not vowels.

The vowels are a, e, i, o, u.

Sample Input

awesome

Sample Output

['w', 's', 'm']

Use a list comprehension to create the required list of letters and output it.

```
word = input()
```

Summary

As we have seen in the previous lessons, Python supports the following collection types: Lists, Dictionaries, Tuples, Sets.

When to use a dictionary:

- When you need a logical association between a key:value pair.
- When you need fast lookup for your data, based on a custom key.
- When your data is being constantly modified. Remember, dictionaries are mutable.

When to use the other types:

- Use lists if you have a collection of data that does not need random access. Try to choose lists when you need a simple, iterable collection that is modified frequently.
- Use a set if you need uniqueness for the elements.
- Use tuples when your data cannot/should not change.

Many times, a tuple is used in combination with a dictionary, for example, a tuple might represent a key, because it's immutable.

Introduction of Functional Programming

Functional programming is a style of programming that (as the name suggests) is based around functions.

A key part of functional programming is higher-order functions.

Higher-order functions take other functions as arguments, or return them as results.

Example:

```
def apply_twice(func, arg):
  return func(func(arg))
def add_five(x):
  return x + 5
print(apply_twice(add_five, 10))
```

The function apply_twice takes another function as its argument, and calls it twice inside its body.

Introduction of Functional Programming

What is the output of this code?

```
def test(func, arg):
  return func(func(arg))
def mult(x):
  return x * x
print(test(mult, 2))
```

Functional programming seeks to use pure functions.

Pure functions have no side effects, and return a value that depends only on their arguments.

This is how functions in math work: for example, the cos(x) will, for the same value of x, always return the same result.

Introduction of Functional Programming

Below are examples of pure and impure functions.

Pure function:

```
def pure_function(x, y):
  temp = x + 2*y
  return temp / (2*x + y)
```

Impure function:

```
some_list = []
def impure(arg):
  some_list.append(arg)
```

Introduction of Functional Programming

The function above is not pure, because it changed the state of some_list.

Is this a pure function?

```
def func(x):
  y = x**2
  z = x + y
  return z
```

Introduction of Functional Programming

Using pure functions has both advantages and disadvantages.

Pure functions are:

- easier to reason about and test.
- more efficient. Once the function has been evaluated for an input, the result can be stored and referred to the next time the function of that input is needed, reducing the number of times the function is called. This is called memorization.
- easier to run in parallel.

Pure functions are more difficult to write in some situations.

Lambdas

Creating a function normally (using def) assigns it to a variable with its name automatically.

Python allows us to create functions on-the-fly, provided that they are created using lambda syntax.

This approach is most commonly used when passing a simple function as an argument to another function.

The syntax is shown in the next example and consists of the lambda keyword followed by a list of arguments, a colon,

and the expression to evaluate and return.

```
def my_func(f, arg):
  return f(arg)
my_func(lambda x: 2*x*x, 5)
```

Functions created using the lambda syntax are known as anonymous.

Lambdas

Lambda functions aren't as powerful as named functions.

They can only do things that require a single expression -- usually equivalent to a single line of code.

Example:

```
#named function
def polynomial(x):
    return x**2 + 5*x + 4
print(polynomial(-4))
#lambda
print((lambda x: x**2 + 5*x + 4) (-4))
```

In the code above, we created an anonymous function on the fly and called it with an argument.

Task

You are given code that should calculate the corresponding percentage of a price.

Somebody wrote a lambda function to accomplish that, however the lambda is wrong.

Fix the code to output the given percentage of the price.

Sample Input

50

10

Sample Output

5.0

The first input is the price, while the second input is the percentage we need to calculate: 10% of 50 is 5.0.

```
price = int(input())

perc = int(input())

res = (lambda x,y:x-y)(price, perc)

print(res)
```

map & filter

The built-in functions map and filter are very useful higher-order functions that operate on lists (or similar objects called iterables).

The function map takes a function and an iterable as arguments, and returns a new iterable with the function applied to each argument.

Example:

```
def add_five(x):
    return x + 5
nums = [11, 22, 33, 44, 55]
result = list(map(add_five, nums))
print(result)
```

map & filter

We could have achieved the same result more easily by using lambda syntax.

```
nums = [11, 22, 33, 44, 55]
result = list(map(lambda x: x+5, nums))
print(result)
```

To convert the result into a list, we used list explicitly.

map & filter

The function filter filters an iterable by leaving only the items that match a condition (also called a predicate).

Example:

```
nums = [11, 22, 33, 44, 55]

res = list(filter(lambda x: x%2==0, nums))

print(res)
```

Like map, the result has to be explicitly converted to a list if you want to print it.

Task

You work on a payroll program.

Given a list of salaries, you need to take the bonus everybody is getting as input and increase all the salaries by that amount.

Output the resulting list.

You can use the map() function to increase all the values of the list.

```
salaries = [2000, 1800, 3100, 4400, 1500]

bonus = int(input())

print(salaries)
```

Generators

Generators are a type of iterable, like lists or tuples. Unlike lists, they don't allow indexing with arbitrary indices, but they can still be iterated through with for loops.

They can be created using functions and the yield statement.

Example:

```
def countdown():
  i=5
  while i > 0:
    yield i
    i -= 1
for i in countdown():
  print(i)
```

Generators

The yield statement is used to define a generator,

replacing the return of a function to provide a result to its caller without destroying local variables.

Due to the fact that they yield one item at a time, generators don't have the memory restrictions of lists.

In fact, they can be infinite!

```
def infinite_sevens():
  while True:
    yield 7
for i in infinite_sevens():
  print(i)
```

Generators

Result:

```
>>>
7
7
7
7
7
7
7
...
```

Generators

In short, generators allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

Finite generators can be converted into lists by passing them as arguments to the list function.

```
def  numbers(x):
  for i in range(x):
    if  i % 2 == 0:
      yield i
print(list(numbers(11)))
```

Generators

Using generators results in improved performance, which is the result of the lazy (on demand) generation of values, which translates to lower memory usage.

Furthermore, we do not need to wait until all the elements have been generated before we start to use them.

What is the result of this code?

```
def make_word():
  word = ""
  for ch in "spam":
    word +=ch
    yield word
print(list(make_word()))
```

Task

Finding prime numbers is a common coding interview task.

The given code defines a function isPrime(x), which returns True if x is prime.

You need to create a generator function primeGenerator(), that will take two numbers as arguments,

and use the isPrime() function to output the prime numbers in the given range (between the two arguments).

Sample Input

10

20

Sample Output

[11, 13, 17, 19]

The given code takes the two arguments as input and passes them to the generator function, outputting the result as a list.

```
def isPrime(x):
    if x < 2:
        return False
    elif x == 2:
        return True
    for n in range(2, x):
        if x % n ==0:
            return False
    return True
def primeGenerator(a, b):
    #your code goes here
f = int(input())
t = int(input())
```

Decorators

Decorators provide a way to modify functions using other functions.

This is ideal when you need to extend the functionality of functions that you don't want to modify.

Example:

```
def decor(func):
  def wrap():
    print("============")
    func()
    print("============")
  return wrap
def print_text():
  print("Hello world!")
decorated = decor(print_text)
decorated()
```

Decorators

We defined a function named decor that has a single parameter func.

Inside decor, we defined a nested function named wrap. The wrap function will print a string, then call func(), and print another string.

The decor function returns the wrap function as its result.

We could say that the variable decorated is a decorated version of print_text -- it's print_text plus something.

In fact, if we wrote a useful decorator we might want to replace print_text with the decorated version altogether

so we always got our "plus something" version of print_text.

This is done by re-assigning the variable that contains our function:

```
print_text = decor(print_text)
```

```
print_text()
```

Now print_text corresponds to our decorated version.

Decorators

In our previous example, we decorated our function by replacing the variable containing the function with a wrapped version.

```
def print_text():
  print("Hello world!")
print_text = decor(print_text)
```

This pattern can be used at any time, to wrap any function.

Python provides support to wrap a function in a decorator by pre-pending the function definition with a decorator name and the @ symbol.

Decorators

If we are defining a function we can "decorate" it with the @ symbol like:

```
@decor
def print_text():
  print("Hello world!")
```

This will have the same result as the above code.

A single function can have multiple decorators.

Which statement can be used to achieve the same behavior as

```
my_func = my_dec(my_func)
```

Task

You are working on an invoicing system.

The system has an already defined invoice() function, which takes the invoice number as argument and outputs it.

You need to add a decorator for the invoice() function, that will print the invoice in the following format:

Sample Input

42

Sample Output

```
***
INVOICE #42
***
END OF PAGE
```

The given code takes the invoice number as input and passes it to the invoice() function.

```
#your code goes here


@decor
def invoice(num):
    print("INVOICE #" +num)


invoice(input());
```

Recursion

Recursion is a very important concept in functional programming.

The fundamental part of recursion is self-reference -- functions calling themselves.

It is used to solve problems that can be broken up into easier sub-problems of the same type.

A classic example of a function that is implemented recursively is the factorial function,

which finds the product of all positive integers below a specified number.

For example, 5! (5 factorial) is 5 * 4 * 3 * 2 * 1 (120).

To implement this recursively, notice that 5! = 5 * 4!, 4! = 4 * 3!, 3! = 3 * 2!, and so on. Generally, n! = n * (n-1)!.

Furthermore, 1! = 1. This is known as the base case, as it can be calculated without performing any more factorials.

Recursion

Below is a recursive implementation of the factorial function.

```
def factorial(x):
  if x == 1:
    return 1
  else:
    return x * factorial(x-1)
print(factorial(5))
```

The base case acts as the exit condition of the recursion.

Not adding a base case results in infinite function calls, crashing the program.

Recursion

Recursion can also be indirect. One function can call a second, which calls the first, which calls the second, and so on. This can occur with any number of functions. Example:

```
def is_even(x):
  if x == 0:
    return True
  else:
    return is_odd(x-1)
def is_odd(x):
  return not is_even(x)
print(is_odd(17))
print(is_even(23))
```

Run the code and see how it works!

Recursion

What is the result of this code?

```
def fib(x):
  if x == 0 or x == 1:
    return 1
  else:
    return fib(x-1) + fib(x-2)
print(fib(4))
```

Task

The given code defines a recursive function convert(), which needs to convert its argument from decimal to binary.

However, the code has an error.

Fix the code by adding the base case for the recursion, then take a number from user input and call the convert() function, to output the result.

Sample Input

8

Sample Output

1000

The binary representation of 8 is 1000.

```
def convert(num):
   return (num % 2 + 10 * convert(num // 2))
```

*args and **kwargs

Python allows you to have functions with varying numbers of arguments.

Using *args as a function parameter enables you to pass an arbitrary number of arguments to that function.

The arguments are then accessible as the tuple args in the body of the function.

Example:

```
def function(named_arg, *args):
   print(named_arg)
   print(args)
function(1, 2, 3, 4, 5_
```

The parameter *args must come after the named parameters to a function.

The name args is just a convention; you can choose to use another.

*args and **kwargs

**kwargs (standing for keyword arguments) allows you to handle named arguments that you have not defined in advance.

The keyword arguments return a dictionary in which the keys are the argument names, and the values are the argument values.

Example:

```
def  my_func(x, y=7, *args, **kwargs):
   print(kwargs)
my_func(2, 3, 4, 5, 6, a=7, b=8)
```

a and b are the names of the arguments that we passed to the function call.

The arguments returned by **kwargs are not included in *args.

Classes

The focal point of Object Oriented Programming (OOP) are objects, which are created using classes.

The class describes what the object will be, but is separate from the object itself.

In other words, a class can be described as an object's blueprint, description, or definition.

You can use the same class as a blueprint for creating multiple different objects.

Classes are created using the keyword class and an indented block, which contains class methods (which are functions).

Classes

Below is an example of a simple class and its objects.

```
class Cat:
  def __init__(self, color, legs):
    self.color = color
    self.legs = legs
felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

This code defines a class named Cat, which has two attributes: color and legs.

Then the class is used to create 3 separate objects of that class.

The __init__ method is the most important method in a class.

This is called when an instance (object) of the class is created, using the class name as a function.

Classes

All methods must have self as their first parameter, although it isn't explicitly passed,

Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Within a method definition, self refers to the instance calling the method.

Instances of a class have attributes, which are pieces of data associated with them.

In this example, Cat instances have attributes color and legs.

These can be accessed by putting a dot, and the attribute name after an instance.

In an __init__ method, self.attribute can therefore be used to set the initial value of an instance's attributes.

Classes

Example:

```
class Cat:
  def __init__(self, color, legs):
    self.color = color
    self.legs = legs

felix = Cat("ginger", 4)
print(felix.color)
```

Classes

In the example above, the __init__ method takes two arguments and assigns them to the object's attributes.

The __init__ method is called the class constructor.

Classes can have other methods defined to add functionality to them.

Remember, that all methods must have self as their first parameter.

These methods are accessed using the same dot syntax as attributes.

Classes

Example:

```
class Dog:
  def __init__(self, name, color):
    self.name = name
    self.color = color

  def bark(self):
    print("Woof!")

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Task

You are making a video game! The given code declares a Player class, with its attributes and an intro() method.

Complete the code to take the name and level from user input,

create a Player object with the corresponding values and call the intro() method of that object.

Sample Input

Tony

12

Sample Output

Tony (Level 12)

Use the dot syntax to call the intro() method for the declared object.

```
class Player:
  def __init__(self, name, level):
    self.name = name
    self.level = level

  def intro(self):
    print(self.name + " (Level " + self.level + ")")

#your code goes here
```

Inheritance

Inheritance provides a way to share functionality between classes.

Imagine several classes, Cat, Dog, Rabbit and so on.

Although they may differ in some ways (only Dog might have the method bark),

they are likely to be similar in others (all having the attributes color and name).

This similarity can be expressed by making them all inherit from a superclass Animal, which contains the shared functionality.

To inherit a class from another class, put the superclass name in parentheses after the class name.

Inheritance

Example:

```
class Animal:
  def __init__(self, name, color):
    self.name = name
    self.color = color

class Cat(Animal):
  def purr(self):
    print("Purr...")
class Dog(Animal):
  def bark(self):
    print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

Run the code and see how it works!

Inheritance

A class that inherits from another class is called a subclass.

A class that is inherited from is called a superclass.

If a class inherits from another with the same attributes or methods, it overrides them.

```
class Wolf:
  def __init__(self, name, color):
    self.name = name
    self.color = color

  def bark(self):
    print("Grr...")
```

```
class Dog(Wolf):
  def bark(self):
    print("Woof")

husky = Dog("Max", "grey")
husky.bark()
```

In the example above, Wolf is the superclass, Dog is the subclass.

Inheritance

The function super is a useful inheritance-related function that refers to the parent class.

It can be used to find the method with a certain name in an object's superclass.

Example:

```
class A:
  def spam(self):
    print(1)
class B(A):
  def spam(self):
    print(2)
    super().spam()
B().spam()
```

super().spam() calls the spam method of the superclass.

Task

You are making a drawing application, which has a Shape base class.

The given code defines a Rectangle class, creates a Rectangle object and calls its area() and perimeter() methods.

Do the following to complete the program:

1. Inherit the Rectangle class from Shape.

2. Define the perimeter() method in the Rectangle class, printing the perimeter of the rectangle.

The perimeter is equal to 2*(width+height)

```
class Shape:
    def __init__(self, w, h):
        self.width = w
        self.height = h
    def area(self):
        print(self.width*self.height)
class Rectangle:
    #your code goes here
w = int(input())
h = int(input())
r = Rectangle(w, h)
r.area()
r.perimeter()
```

Magic Methods & Operator Overloading

Magic methods are special methods which have double underscores at the beginning and end of their names.

They are also known as dunders.

So far, the only one we have encountered is __init__, but there are several others.

They are used to create functionality that can't be represented as a normal method.

One common use of them is operator overloading.

This means defining operators for custom classes that allow operators such as + and * to be used on them.

Magic Methods & Operator Overloading

An example magic method is __add__ for +.

```
class Vector2D:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __add__(self, other):
    return Vector2D(self.x + other.x, self.y + other.y)
```

```
first = Vector2D(5, 7)
second = Vector2D(3, 9)
result = first + second
print(result.x)
print(result.y)
```

The __add__ method allows for the definition of a custom behavior for the + operator in our class.

As you can see, it adds the corresponding attributes of the objects and returns a new object, containing the result.

Once it's defined, we can add two objects of the class together.

Magic Methods & Operator Overloading

More magic methods for common operators:

- __sub__ for -
- __mul__ for *
- __truediv__ for /
- __floordiv__ for //
- __mod__ for %
- __pow__ for **
- __and__ for &
- __xor__ for ^
- __or__ for |

Magic Methods & Operator Overloading

The expression x + y is translated into x.__add__(y).

However, if x hasn't implemented __add__, and x and y are of different types, then y.__radd__(x) is called.

There are equivalent r methods for all magic methods just mentioned.

Example:

```
class SpecialString:
  def __init__(self, cont):
    self.cont = cont
  def __truediv__(self, other):
    line = "=" * len(other.cont)
    return "\n".join([self.cont, line, other.cont])
spam = SpecialString("spam")
hello = SpecialString("Hello world!")
print(spam / hello)
```

In the example above, we defined the division operation for our class SpecialString.

Magic Methods & Operator Overloading

Python also provides magic methods for comparisons.

- __lt__ for <
- __le__ for <=
- __eq__ for ==
- __ne__ for !=
- __gt__ for >
- __ge__ for >=

If __ne__ is not implemented, it returns the opposite of __eq__.

There are no other relationships between the other operators.

Magic Methods & Operator Overloading

Example:

```
class SpecialString:
  def __init__(self, cont):
    self.cont = cont
  def __gt__(self, other):
    for index in range(len(other.cont)+1):
      result = other.cont[:index] + ">" + self.cont
      result += ">" + other.cont[index:]
      print(result)
```

```
spam = SpecialString("spam")
eggs = SpecialString("eggs")
spam > eggs
```

As you can see, you can define any custom behavior for the overloaded operators.

Magic Methods & Operator Overloading

There are several magic methods for making classes act like containers.

- __len__ for len()
- __getitem__ for indexing
- __setitem__ for assigning to indexed values
- __delitem__ for deleting indexed values
- __iter__ for iteration over objects (e.g., in for loops)
- __contains__ for in

There are many other magic methods that we won't cover here, such as __call__ for calling objects as functions, and __int__, __str__, and the like, for converting objects to built-in types.

Magic Methods & Operator Overloading

Example:

```
import random
class VagueList:
  def __init__(self, cont):
    self.cont = cont
  def __getitem__(self, index):
    return self.cont[index + random.randint(-1, 1)]
  def __len__(self):
    return random.randint(0, len(self.cont)*2)
```

```
vague_list = VagueList(["A", "B", "C", "D", "E"])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

We have overridden the len() function for the class VagueList to return a random number.

The indexing function also returns a random item in a range from the list, based on the expression.

Task

We are improving our drawing application.

Our application needs to support adding and comparing two Shape objects.

Add the corresponding methods to enable addition + and comparison using the greater than > operator for the Shape class.

The addition should return a new object with the sum of the widths and heights of the operands,\

while the comparison should return the result of comparing the areas of the objects.

The given code creates two Shape objects from user input, outputs the area() of their addition and compares them.

```
class Shape:
  def __init__(self, w, h):
    self.width = w
    self.height = h
  def area(self):
    return self.width*self.height
  #your code goes here
w1 = int(input())
h1 = int(input())
w2 = int(input())
h2 = int(input())
s1 = Shape(w1, h1)
s2 = Shape(w2, h2)
result = s1 + s2
print(result.area())
print(s1 > s2)
```

Data Hiding

A key part of object-oriented programming is encapsulation,

which involves packaging of related variables and functions into a single easy-to-use object -- an instance of a class.

A related concept is data hiding, which states that implementation details of a class should be hidden,

and a clean standard interface be presented for those who want to use the class.

In other programming languages, this is usually done with private methods and attributes,

which block external access to certain methods and attributes in a class.

The Python philosophy is slightly different.

It is often stated as "we are all consenting adults here", meaning that you shouldn't put arbitrary restrictions on accessing parts of a class. Hence there are no ways of enforcing that a method or attribute be strictly private.

However, there are ways to discourage people from accessing parts of a class,

such as by denoting that it is an implementation detail, and should be used at their own risk.

Weakly private methods and attributes have a single underscore at the beginning.

This signals that they are private, and shouldn't be used by external code. However, it is mostly only a convention,

and does not stop external code from accessing them.

Data Hiding

Example:

```
class Queue:
  def __init__(self, contents):
    self._hiddenlist = list(contents)
  def push(self, value):
    self._hiddenlist.insert(0, value)
  def pop(self):
    return self._hiddenlist.pop(-1)
  def __repr__(self):
    return "Queue({})".format(self._hiddenlist)
```

```
queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
print(queue)
print(queue._hiddenlist)
```

In the code above, the attribute _hiddenlist is marked as private, but it can still be accessed in the outside code.

The __repr__ magic method is used for string representation of the instance.

Data Hiding

Strongly private methods and attributes have a double underscore at the beginning of their names.

This causes their names to be mangled, which means that they can't be accessed from outside the class.

The purpose of this isn't to ensure that they are kept private,

but to avoid bugs if there are subclasses that have methods or attributes with the same names.

Name mangled methods can still be accessed externally, but by a different name.

The method __privatemethod of class Spam could be accessed externally with _Spam__privatemethod.

Data Hiding

Example:

```
class Spam:
  __egg = 7
  def print_egg(self):
    print(self.__egg)
s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

Basically, Python protects those members by internally changing the name to include the class name.

Task

We are working on a game. Our Player class has name and private _lives attributes.

The hit() method should decrease the lives of the player by 1. In case the lives equal to 0, it should output "Game Over".

Complete the hit() method to make the program work as expected.

The code creates a Player object and calls its hit() method multiple times.

```
class Player:
    def __init__(self, name, lives):
        self.name = name
        self._lives = lives
    def hit(self):
        #your code goes here
p = Player("Cyberpunk77", 3)
p.hit()
p.hit()
p.hit()
```

Class & Static Methods

Methods of objects we've looked at so far are called by an instance of a class, which is then passed to the self parameter of the method.

Class methods are different -- they are called by a class, which is passed to the cls parameter of the method.

A common use of these are factory methods, which instantiate an instance of a class,

using different parameters than those usually passed to the class constructor.

Class methods are marked with a classmethod decorator.

Data Hiding

Example:

```
class Rectangle:
  def __init__(self, width, height):
    self.width = width
    self.height = height
  def calculate_area(self):
    return self.width * self.height
  @classmethod
  def new_square(cls, side_length):
    return cls(side_length, side_length)
square = Rectangle.new_square(5)
print(square.calculate_area())
```

new_square is a class method and is called on the class, rather than on an instance of the class. It returns a new object of the class cls.

Class & Static Methods

Technically, the parameters self and cls are just conventions; they could be changed to anything else.

However, they are universally followed, so it is wise to stick to using them.

Static methods are similar to class methods, except they don't receive any additional arguments;

they are identical to normal functions that belong to a class.

They are marked with the staticmethod decorator.

Data Hiding

Example:

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings
  @staticmethod
  def validate_topping(topping):
    if topping == "pineapple":
      raise ValueError("No pineapples!")
    else:
      return True
```

```
ingredients = ["cheese", "onions", "spam"]
if all(Pizza.validate_topping(i) for i in ingredients):
  pizza = Pizza(ingredients)
```

Static methods behave like plain functions, except for the fact that you can call them from an instance of the class.

Task

The given code takes 2 numbers as input and calls the static area() method of the Shape class, to output the area of the shape,

which is equal to the height multiplied by the width.

To make the code work, you need to define the Shape class, and the static area() method,

which should return the multiplication of its two arguments.

Use the @staticmethod decorator to define a static method.

```
#your code goes here


w = int(input())

h = int(input())

print(Shape.area(w, h))
```

Properties

Properties provide a way of customizing access to instance attributes.

They are created by putting the property decorator above a method,

which means when the instance attribute with the same name as the method is accessed, the method will be called instead.

One common use of a property is to make an attribute read-only.

Run the code and see how it works!

Example:

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings
  @property
  def pineapple_allowed(self):
    return False
pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

Properties

Properties can also be set by defining setter/getter functions.

The setter function sets the corresponding property's value.

The getter gets the value.

To define a setter, you need to use a decorator of the same name as the property, followed by a dot and the setter keyword.

The same applies to defining getter functions.

Properties

Example:

Run the code and see how it works!

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings
    self._pineapple_allowed = False
  @property
  def pineapple_allowed(self):
    return self._pineapple_allowed
  @pineapple_allowed.setter
  def pineapple_allowed(self, value):
    if value:
      password = input("Enter the password: ")
      if password == "Sw0rdf1sh!":
        self._pineapple_allowed = value
      else:
        raise ValueError("Alert! Intruder!")
pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

Task

We are improving our game and need to add an isAlive property, which returns True if the lives count is greater than 0.

Complete the code by adding the isAlive property.

The code uses a while loop to hit the Player, until its lives count becomes 0, using the isAlive property to make the condition.

```
class Player:
  def __init__(self, name, lives):
    self.name = name
    self._lives = lives
  def hit(self):
    self._lives -= 1
  #your code goes here
p = Player("Cyberpunk77", int(input()))
i = 1
while True:
  p.hit()
  print("Hit # " + str(i))
  i += 1
  if not p.isAlive:
    print("Game Over")
    break
```

Exceptions

You have already seen exceptions in previous code. They occur when something goes wrong, due to incorrect code or input.

When an exception occurs, the program immediately stops.

The following code produces the ZeroDivisionError exception by trying to divide 7 by 0:

num1 = 7

num2 = 0

print(num1/num2)An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program.

Different exceptions are raised for different reasons.

Exceptions

Common exceptions:

- ImportError: an import fails;
- IndexError: a list is indexed with an out-of-range number;
- NameError: an unknown variable is used;
- SyntaxError: the code can't be parsed properly;
- TypeError: a function is called on a value of an inappropriate type;
- ValueError: a function is called on a value of the correct type, but with an inappropriate value.

Python has several other built-in exceptions, such as ZeroDivisionError and OSError.

Third-party libraries also often define their own exceptions.

Exception Handling

When an exception occurs, the program stops executing.

To handle exceptions, and to call code when an exception occurs, you can use a try/except statement.

The try block contains code that might throw an exception. If that exception occurs, the code in the try block stops being executed,

and the code in the except block is run. If no error occurs, the code in the except block doesn't run.

Exception Handling

For example:

```
try:
  num1 = 7
  num2 = 0
  print (num1 / num2)
  print("Done calculation")
except ZeroDivisionError:
  print("An error occurred")
  print("due to zero division")
```

As the code produces a ZeroDivisionError exception, the code in the except block is run.

In the code above, the except statement defines the type of exception to handle (in our case, the ZeroDivisionError).

Exception Handling

What is the output of this code?

```
try:
  variable = 10
  print (10 / 2)
except ZeroDivisionError:
  print("Error")
print("Finished")
```

A try statement can have multiple different except blocks to handle different exceptions.

Exception Handling

Multiple exceptions can also be put into a single except block using parentheses, to have the except block handle all of them.

```
try:
  variable = 10
  print(variable + "hello")
  print(variable / 2)
except ZeroDivisionError:
  print("Divided by zero")
except (ValueError, TypeError):
  print("Error occurred")
```

You can handle as many exceptions in the except statement as you need.

Exception Handling

What is the output of this code?

```
try:
  meaning = 42
  print(meaning / 0)
  print("the meaning of life")
except (ValueError, TypeError):
  print("ValueError or TypeError occurred")
except ZeroDivisionError:
  print("Divided by zero")
```

Exception Handling

An except statement without any exception specified will catch all errors.

These should be used sparingly, as they can catch unexpected errors and hide programming mistakes.

For example:

```
try:
   word = "spam"
   print(word / 0)
except:
   print("An error occurred")
```

Exception handling is particularly useful when dealing with user input.

Task

An ATM machine takes the amount to be withdrawn as input and calls the corresponding withdrawal method.

In case the input is not a number, the machine should output "Please enter a number".

Use exception handling to take a number as input, call the withdraw() method with the input as its argument,

and output "Please enter a number", in case the input is not a number.

A ValueError is raised when you try to convert a non-integer to an integer using int().

```
def withdraw(amount):
    print(str(amount) + " withdrawn!")


#your code goes here
```

finally, else

After a try/except statement, a finally block can follow. It will execute after the try/except block, no matter if an exception occurred or not.

```
try:
  print("Hello")
  print(1 / 0)
except ZeroDivisionError:
  print("Divided by zero")
finally:
  print("This code will run no matter what")
```

The finally block is useful, for example, when working with files and resources:

it can be used to make sure files or resources are closed or released regardless of whether an exception occurs.

finally, else

What is the output of this code?

```
try:
  print(1)
except:
  print(2)
finally:
  print(3)
```

The else statement can also be used with try/except statements.

In this case, the code within it is only executed if no error occurs in the try statement. Example: Run the code and see how it works!

```
try:
   print(1)
except ZeroDivisionError:
   print(2)
else:
   print(3)
try:
   print(1/0)
except ZeroDivisionError:
   print(4)
else:
   print(5)
```

finally, else

What is the sum of the numbers printed by this code?

```
try:
  print(1)
  print(1 + "1" == 2)
  print(2)
except TypeError:
  print(3)
else:
  print(4)
```

Task

You are making a digital menu to order food.

The menu is stored as a list of items.

Your program needs to take the index of the item as input and output the item name.

In case the index is not valid, you should output "Item not found".

In case the index is valid and the item name is output successfully, you should output "Thanks for your order".

Sample Input

2

Sample Output

Cheeseburger

Thanks for your order

Handle the cases when the input is out of range, as well as when it is not a number.

```
menu = ['Fries', 'Sandwich', 'Cheeseburger', 'Coffee', 'Soda']
#your code goes here
```

Raising Exceptions

You can throw (or raise) exceptions when some condition occurs.

For example, when you take user input that needs to be in a specific format, you can throw an exception when it does not meet the requirements.

This is done using the raise statement.

```
num = 102

if num > 100:

  raise ValueError
```

You need to specify the type of the exception raised. In the code above, we raise a ValueError.

Raising Exceptions

How many exceptions will the following code result in?

```
try:
  print(1 / 0)
except ZeroDivisionError:
  raise ValueError
```

Exceptions can be raised with arguments that give detail about them.

For example:

```
name = "123"
raise NameError("Invalid name!")
```

This makes it easier for other developers to understand why you raised the exception.

Task

You are making a program to post tweets. Each tweet must not exceed 42 characters.

Complete the program to raise an exception, in case the length of the tweet is more than 42 characters.

You can use the len() function to calculate the length of the string.

```
tweet = input()
try:
    #your code goes here
except:
    print("Error")
else:
    print("Posted")
```

Opening Files

You can use Python to read and write the contents of files.

This is particularly useful when you need to work with a lot of data that is saved in a file.

For example, in data science and analytics, the data is commonly in CSV (comma-separated values) files.

Let's start by working with text files, as they are the easiest to manipulate.

Before a file can be edited, it must be opened, using the open function.

```
myfile = open("filename.txt")
```

The argument of the open function is the path to the file.

If the file is in the current working directory of the program, you can specify only its name.

Opening Files

You can specify the mode used to open a file by applying a second argument to the open function.

- Sending "r" means open in read mode, which is the default.
- Sending "w" means write mode, for rewriting the contents of a file.
- Sending "a" means append mode, for adding new content to the end of the file.
- Adding "b" to a mode opens it in binary mode, which is used for non-text files (such as image and sound files).

Opening Files

For example:

```
# write mode
open("filename.txt", "w")
# read mode
open("filename.txt", "r")
open("filename.txt")
# binary write mode
open("filename.txt", "wb")
```

Opening Files

You can combine modes, for example, wb from the code above opens the file in binary write mode.

Once a file has been opened and used, you should close it.

This is done with the close method of the file object.

```
file = open("filename.txt", "w")
# do stuff to the file
file.close()
```

We will read/write content to files in the upcoming lessons.

Reading Files

The contents of a file that has been opened in text mode can be read using the read method.

We have created a books.txt file on the server which includes titles of books. Let's read the file and output the content:

```
file = open("/usercode/files/books.txt")

cont = file.read()

print(cont)

file.close()
```

This will print all of the contents of the file.

To read only a certain amount of a file, you can provide the number of bytes to read as an argument to the read function.

Reading Files

Each ASCII character is 1 byte:

```
file = open("/usercode/files/books.txt")
print(file.read(5))
print(file.read(7))
print(file.read())
file.close()
```

This will read the first 5 characters of the file, then the next 7.

Calling the read() method without an argument will return the rest of the file content.

Reading Files

To retrieve each line in a file, you can use the readlines() method to return a list in which each element is a line in the file.

For example:

```
file = open("/usercode/files/books.txt")
for line in file.readlines():
    print(line)
file.close()
```

Reading Files

If you do not need the list for each line, you can simply iterate over the file variable:

```
file = open("/usercode/files/books.txt")
for line in file:
    print(line)
file.close()
```

In the output, the lines are separated by blank lines, as the print function automatically adds a new line at the end of its output.

Task

You need to make a program to read the given number of characters of a file.

Take a number N as input and output the first N characters of the books.txt file.

The given code opens the books.txt file. Use the file object to read the content of the file.

```
file = open("/usercode/files/books.txt")

#your code goes here
```

Writing Files

To write to files you use the write method.

For example:

```
file = open("newfile.txt", "w")

file.write("This has been written to a file")

file.close()
```

This will create a new file called "newfile.txt" and write the content to it.

In case the file already exists, its entire content will be replaced when you open it in write mode using "w".

Writing Files

If you want to add content to an existing file, you can open it using the "a" mode, which stand for "append":

```
file = open("/usercode/files/books.txt", "a")

file.write("\nThe Da Vinci Code")

file.close()
```

This will add a new line with a new book title to the file.

Remember, \n stands for a new line.

Writing Files

The write method returns the number of bytes written to a file, if successful.

```
msg = "Hello world!"
file = open("newfile.txt", "w")
amount_written = file.write(msg)
print(amount_written)
file.close()
```

The code above will write to the file and print the number of bytes written.

To write something other than a string, it needs to be converted to a string first.

Task

Take a number N as input and write the numbers 1 to N to the file "numbers.txt", each number on a separate line.

Sample Input

```
4
```

Sample Output

```
1
2
3
4
```

The given code reads the content of the file and outputs it.

You can use \n to make line breaks.

Do not forget to close the file after writing to it.

```
n = int(input())
file = open("numbers.txt", "w+")
#your code goes here

f = open("numbers.txt", "r")
print(f.read())
f.close()
```

Working with Files

It is good practice to avoid wasting resources by making sure that files are always closed after they have been used.

One way of doing this is to use try and finally.

```
try:
  f = open("/usercode/files/books.txt")
  cont = f.read()
  print(cont)
finally:
 f.close()
```

This ensures that the file is always closed, even if an error occurs.

Working with Files

Will the close() function get called in this code?

```
try:
  f = open("filename.txt")
  print(f.read())
  print(1 / 0)
finally:
  f.close()
```

Working with Files

An alternative way of doing this is by using with statements.

This creates a temporary variable (often called f), which is only accessible in the indented block of the with statement.

```
with open("/usercode/files/books.txt") as f:
  print(f.read())
```

The file is automatically closed at the end of the with statement, even if exceptions occur within it.

Task

You are given a books.txt file, which includes book titles, each on a separate line.

Create a program to output how many words each title contains, in the following format:

```
Line 1: 3 words

Line 2: 5 words

...
```

Make sure to match the above mentioned format in the output.

To count the number of words in a given string, you can use the split() function, or, alternatively, count the number of spaces

(for example, if a string contains 2 spaces, then it contains 3 words).

```
with open("/usercode/files/books.txt") as f:
   #your code goes here
```