Welcome to Bite-Sized Python Programming!

Bite-Sized courses are a fresh take on our most popular content. Presented in concise, fast-paced, 3-minute lessons, they're designed to give you a quick overview of the most popular areas of programming in a short amount of time.

Ready to get started? Let's check out the first lesson!

Prefer a course that takes a more comprehensive approach? Try Python Programming for Beginners.

Welcome to the Course

Thank you for joining this course!

Before we begin, these are the basic requirements for this course:

Prior knowledge	This course requires no previous knowledge of Python.
Programs & setup	

Hi everyone and welcome to our course on Learning Python by Making a Game! In this course, we will learn the core components of the programming language Python. Our language basics tutorials are geared towards teaching you not just **Python syntax**, but also introducing you to **core coding concepts** and doing so in a way that is easy to understand with relevant examples. We try to simulate real world examples in the code that we write to get you thinking about where we would apply each concept in an actual program.

We have quite a lot of content to cover and each section builds on principles learned in the previous ones. Therefore, **it is very important to make sure you understand each section before moving onto the next**. Although we will go over plenty of examples and explain each concept in detail, it is important to **practice each concept** after we cover it. **Repl.it**, an online programming environment, makes it very easy to quickly write and run code examples. We will relate the examples back to game development to provide a practical and relatable way to learn the concepts. Where possible, we try to explain why we do what we do as well as how we do it. And if it takes a little longer than expected to understand a concept, don’t worry about it! Coding can certainly be difficult; the important thing is to **keep trying and try as many examples as possible** until the concept starts to hit home. Above all, try to have fun! Once you learn to make the game that we’ll build at the end, you’ll have the knowledge and skills to create all sorts of games and other programs using Python!

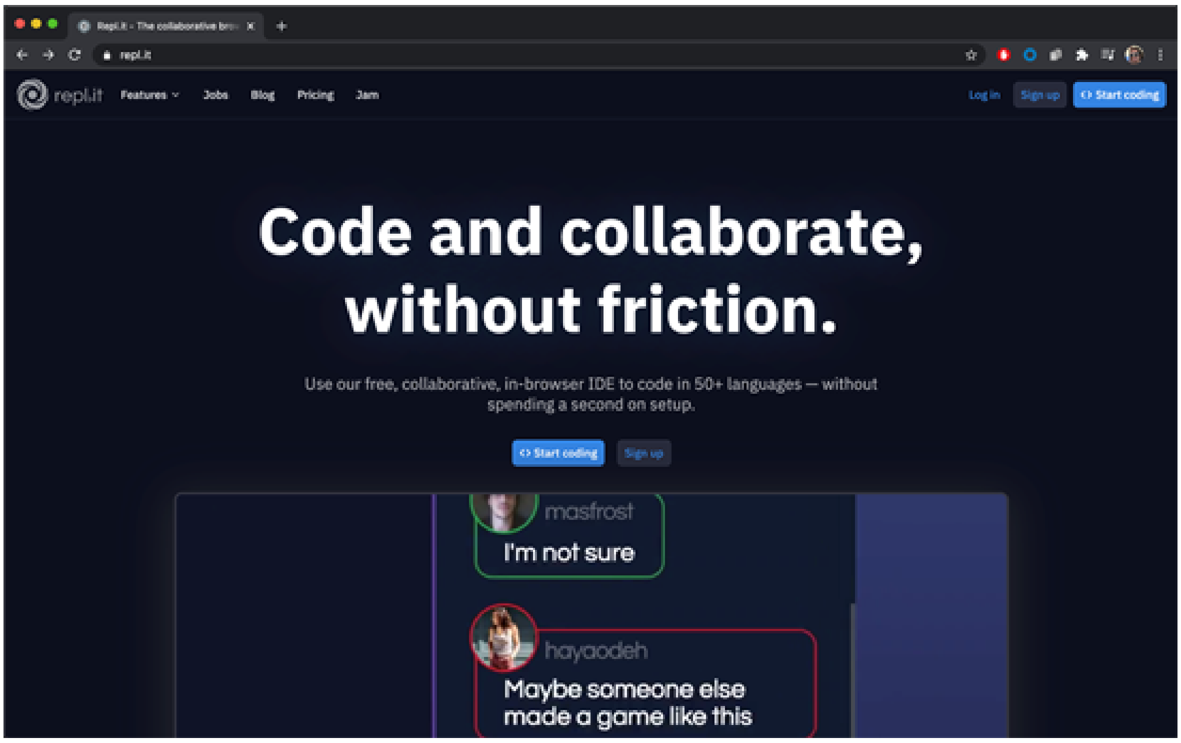
Python is one of the most flexible and dynamic programming languages available. Not only is it easy to learn and to use, but it has a ton of different applications and is used in almost all major areas of programming. The language is considered a **general-purpose language with imperative, functional, and object-oriented paradigms** so already shows tremendous flexibility there. As a result, we see Python used everywhere from web applications, to games, to general desktop applications, particularly in the realm of data science.

Due to its wide variety of applications coupled with **dynamic type system and easy-to-understand syntax**, Python is the choice for programmers around the world, ranking in at the 3rd most used language (as of April 2020). As a result, there are hundreds of libraries, frameworks, SDKs, etc. that extend Python and a huge community of developers willing to answer questions. There is **little to no overhead involved** when writing and running Python programs meaning users can launch right into the more important aspects of developing a program without worrying about the annoying set up. The syntax is clean and concise, allowing beginners to spend more time understanding programming fundamentals without getting caught up in heavy syntax. In fact, the language was built to be as close to spoken or written English as possible. So hopefully Python sounds like an appealing choice and you're excited to learn about it!

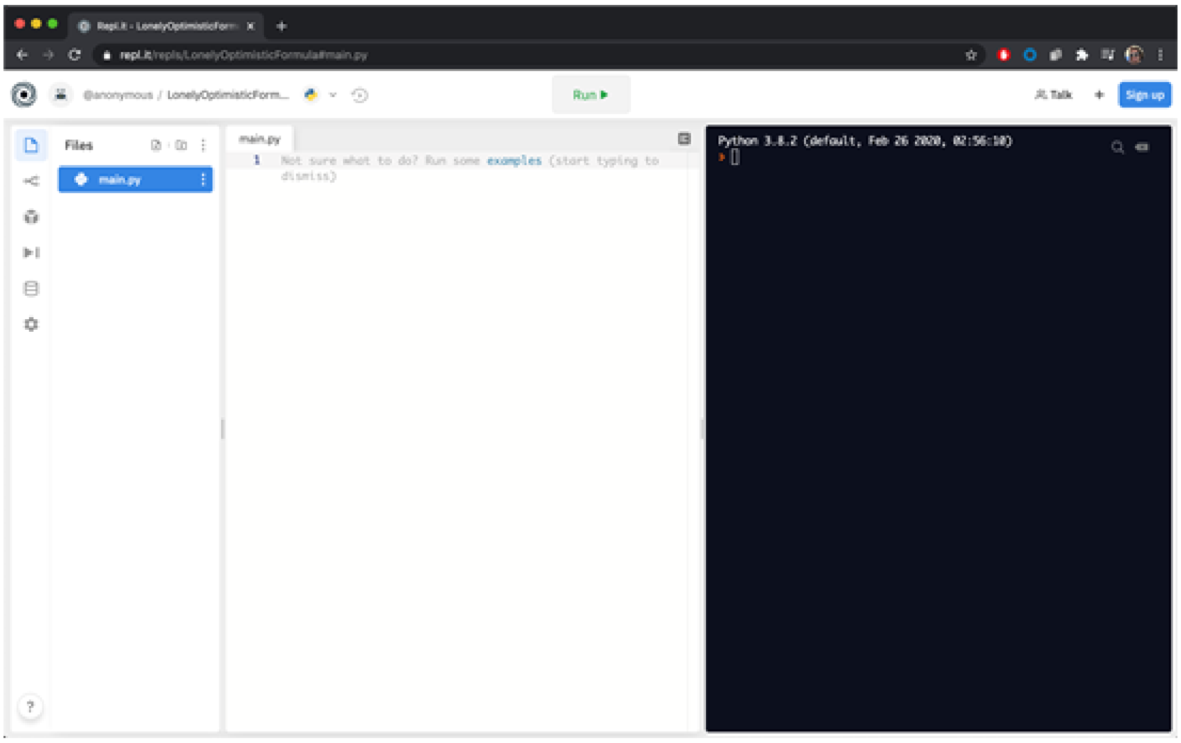
Repl.it is an online programming environment that allows us to write and run code directly in the browser. This is the best way to learn a new programming language or new programming concepts as there is no overhead in downloading languages and IDEs and setting up environments; you just create a new Repl, select the language, and start coding. Repl.it recognizes over 45 different programming languages. When you create an account with Repl.it, you can easily store projects and share code. You can even attach version control and link a Repl to systems like Github. **We will be using Repl.it** to write our language basics code. This way, you can immediately launch into the lessons without having to go through the download and installation process.

Setting up Repl.it

To start, navigate to the website: https://repl.it/


repl.it
Features
Jobs
Blog
Pricing
Jam
Log in
Sign up
Start coding
Code and collaborate, without friction.
Use our free, collaborative, in-browser IDE to code in 50+ languages — without spending a second on setup.
Start coding
Sign up
masfrost
I'm not sure
hayaodeh
Maybe someone else made a game like this

We recommend that you create an account but if you don't want to go through the signup process right now, you can jump right in by selecting "Start Coding", choosing **Python** as the language, and it takes you right to the project page for a new Repl:


Files
main.py
Run
Talk
Sign up
@anonymous / LonelyOptimisticForm...
repl.it/repls/LonelyOptimisticFormula#main.py

Not sure what to do? Run some examples (start typing to dismiss)

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

The window on the left is the file explorer; you can create new files by clicking on the "+" button just to the right of the word "Files" but we will be running everything through the main.py file so there's no need to create another file. **The middle window shows the currently open file** and is where we will write our Python code. **The window on the right is the output** or console window; anything we print and error messages will appear here. To demonstrate how it works, write this in the middle window:

```
print(“Hello world!”)
```

To run the code, press the **"Run" button** just above the middle window. This will print "Hello world!" to the console.

Congratulations on running your first Python program! For now, you can print out any message you want as long as it's between the "" or the '' in the print() function.

Variables are the building blocks on which the rest of the program is written. **They represent data and values** that a program keeps track of and can change as the program runs. Together, variables represent the state of the program. Each variable has **a name, a type, and a value**. There are many different kinds of data that a variable can hold and the type describes exactly that. In Python, we don't explicitly assign a type to a variable; that is interpreted automatically depending on the kind of data it holds and unlike in many other languages, variable types can change in Python. We will look at 4 basic types here: Integers, Floats, Booleans, and Strings.

Integers & Floats

Integers and **Floats** are **numerical types**; Integers are used to store **whole numbers** (can be positive or negative) whereas Floats represent **decimal numbers**. We don't explicitly declare a variable as a Float or an Integer; if we assign a whole number it will be an Integer type and if we assign a decimal number, it will be a Float. To create an integer, we could write this code:

```
x_position = 10
```

This creates a variable called **"x_position"** and assigns the value of **10**. Note the "_" in the name; in Python, we generally name variables and functions with "_" separating the name rather than using camel case. The type of Integer is assigned automatically because we assigned a whole number to the variable. If we want to print out the value of "x_position", we can write this code below it:

```
print(x_position)
```

If you run the code, you should see 10 printed to the console. We can **reassign** the value like this:

```
x_position = 15
```

Which will store a new value (15) in the variable x_position. If you print it out now, you will see 15 printed to the console instead of 10.

We can assign and reassign a Float like this:

```
pi = 3.14
pi = 3.14159
```

We can also represent 15 as a Float by assigning this:

```
x_position = 15.0
```

This actually changes the type of "x_position" **from Integer to Float** because we are now assigning a decimal number to it. We can reassign any data type in Python simply by changing the type of value that the variable holds. If you are ever unsure about the **type of a variable**, we can use the type function like this:

```
x_position_type = type(x_position)
```

You can then print out x_position_type or check its value.

Boolean

The next type of variable we will cover is the **Boolean** type. This represents **True or False values** and can only be **one or the other**. This is very useful for when a property can only have one of two states. For example:

```
is_game_over = False
is_game_over = True
```

Just like before, we can print this variable out by putting it in the print() function. We can also print its type or reassign the type. For example, we can change the value of is_game_over and the Boolean type to an Integer type by doing this:

```
is_game_over = 1
```

Strings

The final data type we will study for now is the **String**. Strings are **any text data** in a program so they could be names, messages, identifiers, etc. as long as the text is between “ “ or ‘ ‘. For example, **the two are equivalent**:

```
name = “Nimish”
name = ‘Nimish’
```

Heads up; you may have to retype the “” if you are copying and pasting over code as the formatting can be different on different platforms. We can even assign string versions of True or False values or even numbers:

```
is_game_over_text = “False”
age_as_a_string = "27"
```

Although these values are **not the same as the literal values of False or 27**. We can also construct strings with other variables inserted into them by using the format() function. For example, if I wanted a String with a name and an age but I didn’t know the age, I could insert the age variable into the String by doing this:

```
age = 27
name_and_age = “Nimish: {}”.format(age)
```

This **tells the compiler to expect a variable value where the {} is** and so we have to pass in a value to fill that (age). If you put this in a print statement and run it, it prints:

```
“Nimish: 27”
```

As it replaces the {} with the value of the variable “age”.

Until you are comfortable with the 4 types of variables, we recommend practicing creating, assigning and reassigning some variables of each type. Try relating the examples to game development for more practical examples. Also, try printing out variable values and types and watch how the types and values change as the variables are reassigned

Now that we are good at creating variables, let’s start using them. **Operators allow us to programmatically change variable values** rather than reassigning them manually. Seeing as we have already used the assignment operator (=) to assign a value to a variable, we will study 3 other kinds: **arithmetic, comparison, and logical operators**.

Arithmetic Operators

Arithmetic operators are your **typical math operations**: + – * / % // **. The last 3 are **modulus** (returns the remainder of a division), the **floor divisio**n (performs division and discards the remainder) and the **exponentiation operator** (raises a number to the power of another). These are typically used between 2 or more numbers like this:

```
x_position = 10
forward = x_position + 1
backward = x_position - 1
```

These add 1 and subtract one from the value of x_position (10) and store the results in some new variables. The others work in the same way and we can perform these operations on variables or on literal values. The only potentially weird ones are the %, // and ** operators so here are some examples to demonstrate how they work:

```
print(5 % 2) # prints 1 because 5 / 2 = 2 with a remainder of 1
print(5 // 2) # prints 2 because 5 / 2 = 2 with a remainder of 1
print(5 ** 2) # prints 25 because 5^2 = 25
```

Anything after the **“#” symbo**l is considered a **comment**. **Comments are ignored by the compiler** – meaning they aren’t code that is run; they are there for us to read and are useful for temporarily commenting out code we don’t want to run or leaving messages for other coders

We can also **combine arithmetic and assignment operators** to store the results back into the variable. For example, if I wanted to do something like this:

```
x_position = x_position + 1
```

I could short-form it to this:

```
x_position += 1
```

These two statements do exactly the same thing; take the current value of x_position, add 1 to it, and store the new value back into the x_position variable. We can combine arithmetic and assignment operators for any of the arithmetic operators. Go ahead and try an example of each of the operations and print out the results.

We can also use the + operator to **append strings** like this:

```
first_name = "Nimish "
last_name = "Narang"
```

```
print(first_name + last_name)
```

Will print “Nimish Narang”. We can also use the **+= operator to stick strings together**.

Comparison Operators

Comparison operators compare two values and return either **True** or **False**. These are: == != >= > <= <. The first one returns true if the two values are equal and the second returns true if the values are not equal; the rest just return the results of the comparison, always returning either True or False. For example, if we were checking to see if we reached the end of a level in a game, we could do something like this:

```
x_position = 10
end_position = 10
is_at_end = x_position == end_position
```

This stores **True** in “is_at_end” **because the two values are equal**. We can even make the comparison a bit more interesting by adding operators. For example, this checks to see if we’re at or past the halfway point:

```
is_at_halfway = x_position >= end_position / 2
```

Logical Operators

Next comes the **logical operators**. These are ways to **manipulate booleans or add additional test cases** and go between two boolean tests or variables. They are “and”, “or”, and “not” (I put these in quotes to distinguish them from the surrounding text). The **not operator** negates a positive to a negative and vice versa. For example, we can turn a Boolean into its opposite by doing this:

```
not_is_at_end = not is_at_end
```

The **and operator** will return true if both booleans are true and false otherwise and the **or operator** returns true if at least one of the booleans is true. If we wanted to set an “is_game_over” variable based on score and position, we might do something like this;

```
score = 9
is_game_over = score >= 10 and is_at_end
```

In this case, is_game_over = False because although is_at_end = True, score >= 10 = False and both have to be True to return True overall. With the “or” operator, only one needs to be True so this stores True:

```
is_game_over = score >= 10 or is_at_end
```

By now, we should have the hang of using basic variables so we can move onto something slightly more complex. **Collections** offer us the ability to **store multiple values within a single variable**. There are three types that we will study here: **lists, tuples, and dictionaries**.

Lists

We will start with **lists which store values based on their index or position**. We can then access them all at once using the name assigned to the list, one at a time using indexing, or several at a time using slicing. Lists can contain a mix of variables of different types but it's **better practice,** where possible, **to stick to just one type**. Let's say we want to keep track of where our enemies are in a game. We could create a list of all enemy positions like this:

```
enemy_positions = [5, 10, 15]
```

Notice how it is assigned just like a regular variable but with multiple values inside of the [] separated by commas. We can **reassign an entire list** just like any other variable:

```
enemy_positions = [5, 10, 15, 20]
```

If you print out the list, you will see the list of values printed in the same format as declared:

```
print(enemy_positions)
```

Using Length and Indexes

Lists can contain as many elements (values) as we want and can be **empty** if there are no values to store. We can get the length of a list (number of elements in it) by doing this:

```
length = len(enemy_positions)
```

The length variable is of type Integer. If we want to access individual elements, we do so based on their index or position in the list. **Indexing is 0-based** so the first element is at index 0, then 1, then 2, etc. To get an element we call the name of the list with the index of the element between [] like this:

```
last = enemy_positions[3]
```

This just copies the value at index 3 in the list (20) into the variable "last" but changing last has no effect on the list. If we wanted to change a value, we could do this:

```
enemy_positions[0] = 6
```

Slicing

Now the list has changed; the first value has been reassigned to a 6. We can also **retrieve multiple elements through the slicing operator** ([:]). This takes in a lower bound, upper bound, and step but we can include all or only some of these. A simple example would be:

```
first_two = enemy_positions[0:2]
```

The **upper bound is not included in the slice** so this returns a list of just the 0th and 1st elements. If we try to access or change a list element that doesn't exist (for example, an element at the index 10 in our list), we get an error so we always want to **make sure the index we are accessing exists**.

Manipulating List Properties

There are several operations we can perform on lists that help us manipulate them or retrieve properties. We won't cover all of them but we will look at **append, insert, del, and remove**. We can add an item to the end with the **append** function like this:

```
enemy_positions.append(25)
```

We can **insert** an item at a **specific index** by doing this:

```
enemy_positions.insert(1,9)
```

Which inserts the value of 9 at the index 1 and shifts the proceeding elements up the list by index 1. We can also **remove** a specific item by doing this:

```
enemy_positions.remove(6)
```

This will remove the value if it exists in the list and shift the other elements to close the gap. We can **remove an item at a specific index** by doing this:

```
del(enemy_positions[2])
```

We recommend checking trying each of these functions out and taking a look at some of the other available functions; generally they are highly efficient and simplified.

Tuples

The next type of collection we will study is the **tuple**. This acts very **similar to a list but is immutable** meaning we **cannot add** or **remove** elements or **change** the elements themselves; we use tuples for storing values only. For example, we could record a high score like this:

```
high_score = ("Nimish", 120)
```

Note how we are using multiple data types although all elements can be of the same type if we want. We can **reassign the entire tuple** by assigning a new value to the variable like this:

```
high_score = ("Holly", 150)
```

However, we **cannot modify individual elements by index**; we can only fetch the values:

```
# high_score[0] = "fadsfsd" - not allowed so commented out
name = high_score[0]
```

This restricts the types of functions we can call on tuples but at least we can get the length like this:

```
length = len(high_score)
```

And we can **check to see if values exist** in tuples or lists by using the in operator:

```
does_contain = "Holly" in high_score
```

This stores True or False depending on whether the tuple contains “Holly”. We can also apply a lot of this logic to Strings. For example, we can fetch a single character from a String by index, a substring of characters by index range, check to see if a String contains a substring, and get the length of a String:

```
name = “Holly”
first = name[0]
first_two = name[0:2]
is_in = "Hol" in name
length = len(name)
```

Dictionaries

The final collection type we will look at is the **dictionary**. **Dictionaries attach their values to keys** rather than storing them at specific positions or indexes. They act similar to hashmaps in other languages. The benefit of using a dictionary is that if we need to access a specific element, **we don't need to know its position** (as we would have to if we stored it in a list or tuple); we only need the **key** under which we stored it. We can set up a dictionary like this:

```
actions = {"r":1, "l":-1}
```

You can see that there are two key-value pairs where the keys are strings and the values are integers. The **keys and values can be whatever types you want** but string-int makes sense for us. To access a value, **we need the key** and do so in two ways:

```
right = actions["r"]
right = actions.get(“r”)
```

Both store 1 because the value at the key “r” is 1. The difference is that “.get()” is safer and will return “None” if the key does not exist whereas [value] will just crash the program if the key does not exist.

Manipulating Dictionary Data

Similarly, **we can change values via their key and if the key doesn't exist, the dictionary inserts a new key and stores the value**. For example:

```
actions["r"] = 2
```

Changes the valued stored under “r” to 2 instead of 1 but:

```
actions["u"] = 1
```

Creates a **new key** “u” because it didn't exist before and stores the value 1. There are dictionary specific functions that we can access in the same way as lists and tuples. For example, we can get **a list of (key, value) tuples**, **a list of just the values**, or **a list of just the keys** like this:

```
items = actions.items()
values = actions.values()
keys = actions.keys()
```

We can **remove** a key-value pair by doing one of these two:

```
del(actions["u"])
actions.pop("r")
```

We can also check to see if a key (and therefore some value) exists like this:

```
is_in = “l” in actions
```

Now that we're familiar with some of the building blocks, let's introduce some logic and decision making into our code via **control flow**. This allows us to **test conditions and execute code if they evaluate to True**. By incorporating control flow, we can choose to execute certain parts of the code only if certain conditions are met or execute code multiple times with loops.

If Statements

The simplest form of this is the **if statement** which is a way to e**xecute code only if some test returns true**. The test is a boolean (True or False) and is **often just comparing some variable value to another**. For example, to build a movement control system, we typically take in a user, test what it is, and provide different functionality for each keystroke. Let's represent a keystroke with a string ("r" or "l" for right and left respectively) and if it's anything else, we will print "invalid key". We can test for just "r" like this:

```
key = "r"

if key == "r":
  print("move right")

print("done")
```

This will print "move right" to the console followed by "done" ("done" is printed regardless because it is not part of the if-statement so is run when the if statement finishes running). Note the indentation; **Python does away with () or {} in its control flow and instead uses a : to signify the end** of the test and indentation to signify that statements are part of the if statement body. We only have one test here but we also want to test for the "l" key so we can add an **elif**:

```
key = "r"

if key == "r":
  print("move right")
elif key == "l":
  print("move left")

print("done")
```

The **elif statement is only run if the previous tests return False** so if key = "l", the first test returns False so we skip the "print("move right")" code and run the elif statement test. If we want some **default code to execute if all previous tests return False**, we can use an else clause:

```
key = "r"

if key == "r":
  print("move right")
elif key == "l":
  print("move left")
else:
  print("invalid key")

print("done")
```

This will only print “invalid key” if all previous tests fail. **If any one test passes, we ignore the rest of the test cases** and move onto the next code (print(“done”)).

We can use pretty much any combination of if-elif-else such as just if-else or multiple elif statements. We can also nest if statements within other if statement bodies or make the conditional tests more complex by adding logical “and” or “or” operators.

Now that we can perform tests and execute code once, the next step is to be able to perform tests and execute code multiple times as long as some condition continues to be true. For this, we use loops. **Loops provide a way to execute code multiple times** without having to type the same code again and again.

While Loops

The simplest form is the **while loop** and it acts very much **like an if statement run multiple times**. It performs a test and executes code if the test returns True but instead of executing the code once, **it continuously executes the code until the test eventually returns False**. This can be dangerous because if the test never returns False, the loop will run infinitely and the program will crash so we need to make sure that we build the loop in such a way that it will eventually exit.

Games and other programs at their heart, often contain a **big game loop** that continuously performs checks, handles player interaction, renders graphics, etc. as long as the game is still running and breaks when the user does something to end the game. We can build a very simplified version of this by moving a player as long as they are not quite at the end position and then exiting the loop when they are:

```
position = 0
end_position = 10

while position < end_position:
  position += 1
  print(position)
```

Again, note the : and the indentation. This will increase the position by 1 for each loop iteration and print the new position out so it runs 10 times. Once the position evaluates to 10 (as it is updated with each iteration), the condition position < end_position evaluates to False and so the loop breaks or stops running and any proceeding code is executed.

We can also control the loop iterations through the use of “continue” and “break” statements. **Continue will force the loop to go onto the next iteration** (and skip any code left for that loop cycle) whereas **break will exit the loop immediately**. For example, if we wanted to stop moving the player forward if they collide with an enemy, we could do something like this:

```
position = 0
end_position = 10
enemy_position = 5

while position < end_position:
  position += 1
  print(position)
  if position == enemy_position:
    print("Game over!")
    break
```

In this case, The loop will only run 5 times because when the position = 5, position == enemy_position = True and the break statement is reached, causing the loop to stop running.

For Loops

The next type of loop is the **for loop**. This acts similar to a while loop but it **runs a preset number of times**. **We build right into the loop when to start and when to stop** so we know how many times it will run. We can **pair it with a collection type to visit every element** of a list. When we pair it with a list, for example, we iterate through the list with each iteration of the loop visiting the next element of the list. We can visit every member of a list and print it out with a for loop like this:

```
enemy_positions = [5, 10, 15]

for enemy_position in enemy_positions:
  print(enemy_position)
```

Note again, the indentation. The for in loop starts at the beginning of enemy_positions (index 0). It assigns the value of enemy_positions[0] into enemy_position and runs the loop body (print(enemy_position)). Then it moves to the next iteration and visits the next member of the list (index 1). This process continues, each time updating the enemy_position to contain the value of the current index of enemy_positions until it has run every element in the list. If the list is **empty,** the loop will run **0 times**.

We can also **pair for loops with ranges** if we just want them to run a set number of times. For example, if we wanted a loop to print “Hello” 5 times, we could do so like this:

```
for i in range(0,5):
  print("Hello")
```

We don’t really care so much about i, we just want the loop to run 5 times so it runs as many times as there are elements in the range (the upper bound, 5, is not included in the range so the range is [0,1,2,3,4]). The final thing about loops is that **every for loop can be turned into a while loop but not every while loop can be turned into a for loop** because sometimes we don’t know how many times a while loop will run but we always know how many times a for loop will run.

Now onto functions. **Functions are just contained blocks of code that run only when we choose**. They remain dormant until we call on the function, at which point the code within the function is run. The benefit of doing this is that we can **hide away code so that it is run only when we want it to run**. This also allows us to **execute the same functionality** at potentially many different points in the program without having to rewrite the code; we can simply call upon the function and it runs the code in its body.

Basic Functions

A simple example of this might be a move function. We would use this to change a character's position in a game but would only want to do so when the user presses the right button. We could create an external position variable, and just increase it when the function is called like this:

```
position = 0

def move_player():
  global position
  position += 1
  print(position)
```

We start by creating the position variable to keep track of the position as the program runs. We then open up the function with the **def keyword** which starts a function definition. The function needs a **name** (move_player) and t**he empty parentheses beside it indicate that it does not take in any inputs**. We have to call "global position" to get access to the position variable as it was declared outside of the function body. Then we can increase the value of pos by 1 and that's it!

However, if we run this code, the position variable does not change. That's because we have defined the function but we haven't actually run the code yet. The code within a function is **not run until we call it**. We call upon a function by calling the name and providing any necessary inputs within the parentheses, in our case, like this:

```
move_player()
```

If we add that line after the function def and then print the value of position, we should see that position = 1 as we have called the function which runs the code within it. Before moving onto parameters and return values, note that if we create a variable within a function, we can **only access the variable within that function as variables only exist within the function/control flow/class in which they're declared**.

Parameters

Our previous function from last lesson could definitely be improved. For starters, we might want to move forwards by more than 1 or even backwards. We could do that by taking in a movement amount and using that to change position. Functions can **take values in via parameters which we place within the ( )** and can then use like regular variables. For example, we could take in a movement amount and use it like this:

```
position = 0

def move_player(by_amount):
  global position
  position += by_amount
  print(position)
```

Now when we call the function, it needs an **input value** for by_amount so we can do something like this:

```
move_player(5)
```

To move forwards by 5. We can pass in as many parameters we want and they can be of whatever type we want. However, **we have to pass in at least one value for each parameter** when calling a function.

Return Values

Now if we wanted to do away with the global variable (which is better practice; **we should avoid global variables if possible**), we can take in the current position and the movement amount like this:

```
position = 0

def move_player(position, by_amount):
  position += by_amount
  print(position)
```

We then call the function like this:

```
move_player(position, 5)
```

Note that this updates the value of position within the function and prints out the correct value but if we print(position) outside of the function, its value is still 0. That is because the value is just copied when passed in as a parameter (and we are not using the global variable position). To fix this, we can use return values. **Return values act as output from a function** and can be captured when calling a function like this:

```
position = 0
```

```
def move_player(position, by_amount):
  position += by_amount
  return position

position = move_player(position, 5)
```

Now we will see the value of position updated to 5 outside of the function. We can also use return statements to break out of functions prematurely just like with break statements. If a function does not need to output a value but we want to break out of it prematurely, simply use a **“return” with no value beside it**.

Now let’s put together everything we have learned in this final big topic. **Objects are code entities with state and behavior**. The **state** is represented by variables that make up various **properties** or **attributes**. This allows us to create complex data types with many values attached to them. The **behavior** is managed by functions that the object can run; typically, these functions modify the state in some way.

Classes act as blueprints for objects, specifying what variables an object should use to make up its state and also defining any functions the object might want to run. We can then create instances of these classes which are the objects themselves. We can think of this a bit like function construction where the class is the function definition as it describes what the object can do and the object is like calling the function because we are actually running some code found within the class. Classes also use **special initializer functions** to create instances and set up the initial state of the object.

Fundamentals

An example of this might be an object in a game. Every object will at the very least have x and y positions and maybe a name. Although a real game object will have more attributes, this is enough for our demonstration. We can create a class to represent the object like this:

```
class GameObject:

  def __init__(self, name, x_pos, y_pos):
    self.name = name
    self.x_pos = x_pos
    self.y_pos = y_pos
```

Let’s break this down. We are creating a **class** called **GameObject** that has three properties: **name, x_pos, and y_pos**. We are setting the initial values of these in the **__init__ function** (note that we have **two underscores** before and after ‘init’). This is just a **special function that takes in parameters for the initial values** for the three properties and sets them up. The **“self”** keyword is used to access the variable or function that belongs to a GameObject object. When we call a function, we don’t need to pass in a value for “self” but for now, every function that we create belonging to a class has to have the self parameter.

Much like a function, creating this class defines behavior that we can run but nothing really happens. To see this class in action, we can create an instance of it like this:

```
game_object = GameObject("Enemy", 1, 2)
```

Note that we can use the initializer implicitly; it is a special function so we just need to call the class name (GameObject) and pass in the values for the parameters for the initializer. Now the game_object variable is of type GameObject and contains the three variables name, x_pos, and y_pos. We can access them with . syntax and even set them like this:

```
name = game_object.name
game_object.name = "Enemy 1"
```

From now onward, the game_object value of name = “Enemy 1” instead of “Enemy”. We can do the same thing for x_pos and y_pos. We can create many **instances** of this class, all with unique values for name, x_pos, and y_pos:

```
other_game_object = GameObject("Player", 2, 0)
```

other_game_object has all different values compared to game_object.

Functions in Classes

Classes can **also contain functions** that objects can call. If we wanted a move function, we can add it to the class like this:

```
class GameObject:

  def __init__(self, name, x_pos, y_pos):
    self.name = name
    self.x_pos = x_pos
    self.y_pos = y_pos


  def move(self, by_x_amount, by_y_amount):
    self.x_pos += by_x_amount
    self.y_pos += by_y_amount
```

This will update the x and y positions based on values passed in when called. Note again the “self” parameter and the fact that we are setting self.x_pos and self.y_pos. If we called x_pos or y_pos without the self, **we are no longer referring to the variables that belong to the class GameObject**. To use this function, we create an instance and use the **. syntax** just like with variables:

```
game_object = GameObject("Enemy", 1, 2)
game_object.move(5, 10)
```

After calling game_object.move(), the x_pos and y_pos stored in game_object will change. You can verify any of these by printing out the values.

About References

Objects are reference types which means that upon assignment, we maintain a reference to a specific object rather than just copying some values. So far, all of the variables (Integers, Floats, Booleans, Strings, Lists, Dictionaries, etc.) are all value types meaning their values are copied during assignment. Take this example:

```
one_int = 5
another_int = one_int
```

Here we created a variable called one_int and assigned the value of 5. We created another variable (another_int) and assigned it whatever value one_int contained. This is simply copying the value so another_int = 5. If I change one of them:

```
another_int = 10
```

The other is unaffected. Here, one_int = 5 but another_int = 10. This is because these are both value types and assigning one equal to the other is just copying the current value. However, with objects,

we create a reference when assigning one to the other. Take this example:

```
game_object = GameObject("Enemy", 1, 2)
other_game_object = game_object
```

Here, other_game_object is a reference to game_object so they are actually both referring to the same object rather than copying values, therefore the values are shared between them both. This means that whenever we change a value within one of them, we also change the other:

```
other_game_object.name = "new name"
```

Changes the name of both other_game_object and game_object to “new name”. This can lead to unexpected problems if not properly managed.

Inheritance

Another feature of object orientation is inheritance. **Inheritance allows one class to “inherit” or essentially copy over all of the variables and functions of another class** as well as implement its own. This is very useful when we want a set of variables and functions that many different classes will share. For example, a non-player character in a game will likely have a lot of the same attributes and functionality that a player character will (position, size, name, etc.) so rather than rewriting the same setup code for both, we can create “superclass” and have player character and non-player character “subclasses” inherit from it. Here is an example of that:

```
class GameObject:

  def __init__(self, name, x_pos, y_pos):
    self.name = name
    self.x_pos = x_pos
    self.y_pos = y_pos


  def move(self, by_x_amount, by_y_amount):
    self.x_pos += by_x_amount
    self.y_pos += by_y_amount


class Enemy(GameObject):

  def __init__(self, name, x_pos, y_pos, health):
    super().__init__(name, x_pos, y_pos)
    self.health = health


  def take_damage(self, amount):
    self.health -= amount
```

Here we start with our classic GameObject class. This is the **superclass** because it **acts like a base class implementing generic functionality**. The second class is an Enemy, a subclass of the GameObject class (denoted by the Enemy(GameObject)) that has all of the same attributes (name, x_pos, y_pos) and functions (move()) that the GameObject class has but can also define its own unique variables and functions (health, take_damage()). The initializer of the Enemy class is calling upon the superclass’s initializer in this line:

```
super().__init__(name, x_pos, y_pos)
```

When run, this executes the code found in the **GameObject initializer** to set up the name, x_pos, and y_pos. We are also setting up health which is why we are defining a new initializer in Enemy. To create an Enemy we can do this:

```
enemy = Enemy("Enemy", 5, 10, 100)
```

Just like before, we can access variables and functions unique to Enemy or those shared between

GameObject and Enemy. The restriction here is that if we create an instance of GameObject:

```
game_object = GameObject("Game object", 1, 2)
```

This has no knowledge of health or take_damage() as it is the superclass and that variable and that function were defined in the subclass. When used properly inheritance is a very powerful tool that makes code maintainability much easier and helps us to minimize the amount of code we have to write.

That’s it! Congratulations for reaching the end of our Python language basics course! You now know enough to begin writing your own Python programs. We learned the fundamentals of Python covering first variables, then operators, collections, control flow, functions, and finally objects and classes. From here, we recommend going over each topic again and coming with new examples of where we would use each of them in a real program. Once you feel really comfortable with each topic, try putting together a complete Python program to accomplish a simple task and keep building from there. Python also has a framework for just about any area of programming so it’s worth seeing which ones are popular and reading up on the documentation. We look forward to seeing what sorts of projects you all build and hope to see you in future courses!

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

main.py

Found in the Project root folder

This Python code is a broad overview of fundamental coding concepts including variables, data types (numbers, booleans, strings, lists, tuples, dictionaries), operators (arithmetic, comparison, assignment), control structures (if statements, while and for loops), functions, and classes. The latter extensions of the code introduce object-oriented programming via classes, objects, and inheritance and create simple game object instances like enemies with specific behaviors.

```
# 2. Intro to Repl.it
print("Hello world!")

# 3. Numbers and booleans
position = 27
pi = 3.14
is_game_over = False
print(position)
print(type(position))

# 4. Strings
hello_text = "Hello world!"
name = 'Nimish'
print(hello_text)
print(len(name))

# 5. Arithmetic and assignment operators
# Operators: = + - * / % // **
position = 5
forwards = position + 1
backwards = position - 1
position += 1
print(position)

# 6. Comparison operators and logical
# Operators: > >= < <= == != not or and
is_at_end = position == 10
is_past_halfway = position >= 5
is_game_over = not is_game_over
points = 10
is_game_over = points >= 10 and is_at_end
print(is_game_over)

# 7. Lists
enemy_positions = [5, 10, 15]
first_position = enemy_positions[0]
enemy_positions[1] = 12
enemy_positions.append(20)
```

```
del(enemy_positions[1])
print(len(enemy_positions))
print(enemy_positions)

# 8. Tuples
high_score = ("Nimish", 100)
high_score = ("Holly", 120)
print(len(high_score))

# 9. Dictionaries
actions = {"r": 1, "l": -1}
right = actions["r"]
actions["u"] = 1
print(actions)

# 10. If statements
action = "r"
if action == "r":
  print("Move right")
elif action == "l":
  print("Move left")
else:
  print("Invalid key")

# 11. While loops
position = 0
end_position = 5
while position < end_position:
  print(position)
  position += 1
print("You've reached the end!")

# 12. For in loops
for action in actions:
  print(action)

# 13. Functions
position = 0
def move_player():
  global position
  position += 1
move_player()
print(position)

# 14. Function parameters
def move_player(by_amount):
  global position
  position += by_amount
move_player(5)
print(position)

# 15. Return values
def move_player(position, by_amount):
  return position + by_amount
position = move_player(0, 5)
```

```
# 16. Intro to classes and objects
class GameObject:

  def __init__(self, name, x_pos, y_pos):
    self.name = name
    self.x_pos = x_pos
    self.y_pos = y_pos

  def move(self, by_x_amount, by_y_amount):
    self.x_pos += by_x_amount
    self.y_pos += by_y_amount

game_object = GameObject("Enemy", 1, 2)
game_object.move(5, 6)
print(game_object.name)

# 17. Inheritance:
class Enemy(GameObject):

  def __init__(self, name, x_pos, y_pos, health):
    super().__init__(name, x_pos, y_pos)
    self.health = health

  def take_damage(self, amount):
    self.health -= amount
    if self.health < 0:
      return

enemy = Enemy("Enemy 1", 5, 10, 100)
enemy.take_damage(20)
print(enemy.health)
```