PYTHON READER

Important note: This PDF is static! As the course goes on, the textbook is bound to change and the latest version is available at https://codeinplace.stanford.edu/cip3/textbook/welcome.

Last Updated: June 5th, 2023

TABLE OF CONTENT

— Intro to Python —

Welcome to Python

Quest

Greetings, eager traveler! Your adventures in programming have led you to the foot of a tall, beautiful mountain! Today marks the beginning of your journey through Python, a journey that will inspire you and teach you so many wonderful things about computers and how they work. We had such a good time playing with Karel, and now it's time to say goodbye to our old friend and say hello to Python, a far more powerful and even more exciting programming language! Python is used everyday by professional programmers to build amazing new things, and you get to learn how it works! Python may seem new, but you'll quickly see how familiar it feels having worked with Karel. So, are you ready? Let's look at our first Python program!

A Blank Python Program

Before you get started writing anything, there are a few things you must add to every Python program you write. Here is a template to start writing your program:

```
def main():
    # Your code goes here

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

Every Python program in this class must have a main function. We'll explain what a function is soon and how to create one of your own. Starting out, though, you will define the main function with the line `def main():` and your code will go "inside" the main function by indenting every line of your code once. The stuff at the very bottom is just some required code to tell the computer what it should do when you run the program. It's really just pointing the computer to where the first line of code is. You'll just have to memorize this template going forward, though you can always come back here later to refresh yourself if you need.

High-Low

This is Python code for a game called High-Low. The computer picks a secret number and you try to guess it. After each guess, the computer tells you whether you were too high or too low.

Note: The secret number will always be a whole number, never a decimal. If you guess anything that's not a number, like a word or a weird symbol, the program might break. If that happens, just run it again to reset everything.

```python
import random

def main():
    # Program selects a new random number
    secret_number = random.randint(1,100)

    print("I am thinking of a number from 1 to 100. What number am I thinking of?")
    guess_str = input()
    guess = int(guess_str)

    while guess != secret_number:
        if guess > secret_number:
            print("Nope! That guess was too high!")
        if guess < secret_number:
            print("Sorry! That guess was too low!")
        print("Take another guess!")
        guess_str = input()
        guess = int(guess_str)
    print("You guessed my number! Way to go!")


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

Believe it or not, by the end of this section, *you* will be able to write this program yourself. It may look like a lot right now, but we are going to go through everything used in this program step by step! Notice this program still follows our template from earlier. Even complex programs like this one still need a main function.

If you're feeling lost right now, don't worry. It's ok if none of this makes any sense to you yet; we are here to learn! You got this! Continue learning below with our first lesson: commenting!

Commenting

Can you find the line that starts with a hashtag in the High-Low program above?

```python
# Program selects a new random number
```

As you write longer and more complicated programs, it will get much harder to understand exactly what your program is doing just by looking at it. It can be helpful to leave little notes for yourself or anyone else reading your code to tell the reader what the program does. Comments are a way to write bits of descriptive text in your program that Python will ignore when running your program. There are two ways to add comments to your code:

Single-Line Commenting: Using the # symbol begins a single-line comment. Anything written after the # on the same line will not be interpreted as code and is skipped when the program is run.

Block Commenting: If you want to write a longer comment that will use multiple lines, it would be a pain to add a # for each line you use. Instead, we enclose the comment block with two triple quotes ''' or """ and everything between them gets "commented out" (just make sure to be consistent with using single or double quotes).

```
'''
Nothing in this entire block of text will affect the program.
We can safely include as much text as we want here, even text that looks like code. If I write
something like:
if condition:
    do_something()

This may look like code to you, but Python will still ignore it. Block commenting is very useful if you
want to comment out a section of code without totally erasing it.
'''

if condition: # Only this text will be ignored on this line
    do_something()
    do_something_else()

    """
    This is another block comment, but notice how this one
    is indented. Unlike single-line comments, block
    comments must be appropriately indented if they are
    nested inside other code.
    """
do_this_one_after_the_if_finishes()

# And now the program is done
```

Without all the comments, the only code that actually matters looks like this:

```
if condition:
    do_something()
    do_something_else()
do_this_one_after_the_if_finishes()
```

Comments make your programs much longer, but they are very important for helping readers know what’s going on. Always include them where it is helpful!

Interpreted vs Compiled

You should know that Python treats single-line and block comments differently, but we can't explain why until we've explained a little more about how Python works. Python is what's known as an interpreted language. When you run code in an interpreted language, another program called the interpreter reads your code and tells the computer what to do. This is different from other languages like Java or C++, which are compiled languages. With a compiled language, a compiler translates your code into binary, the language your computer understands. The computer itself then reads that binary and runs it.

Ok, now we can explain commenting! The distinction between single and block comments is that *a single-line comment is fully ignored* by the interpreter, while *a block comment is valid code* that has no effect on the outcome of the program. If the interpreter sees a hashtag, it thinks to itself, "Great, I can skip this part!" With block comments, however, the interpreter doesn't ignore the comment at all; it's just that the comment isn't an instruction per se. The interpreter just reads it as, "Ok, there's a comment here!" and continues on.

So, why should you care? All of this really just means that you need to properly indent your block comments if you nest them inside of something like a loop. If you forget, Python will either yell at you for not indenting correctly, or worse, your code will break. If the block comment is not nested inside of anything, you can safely leave it unindented. This idea is illustrated in the above example. Do you see why we indented one block comment but not the other?

Console

In order to run Python programs, we use something called the console. The console allows you to run programs and see the result of whatever you run.

The console is where you will see error messages, prompts for user input, and output from print statements (user input and print statements will be covered in future sections). To run a program in the console, you write your program name after the keyword python:

```
python filename.py
```

For instance, if our high-low program was stored in a file called high_low.py, we could run it in the console like this:

console

```
$ python high_low.py
# program runs
# results are displayed here below
```

If the program you just ran raises an error, an error message appears in the console explaining what went wrong. Error messages look something look like this:

console

```
$ python high_low.py
Traceback (most recent call last):
File "high_low.py", line 3
SyntaxError: invalid syntax
```

Don't worry about what that error message means yet. For now, we just want to be able to see how an error message would appear in the console.

Important Note: make sure to include the `.py` in the file name and avoid spaces when naming files. Otherwise, you might get an error that has nothing to do with your code.

console

```
$ python high_low
[Errno 2] No such file or directory
```

Print

Quest

Get ready, because you are about to experience a rite of passage for all new programmers: writing your first *Hello World* program! To do this, we will learn about the print function, the first of many of our new tools. Let's go see how it works!

Printing to the Console

The print function, put simply, writes text to your console. The computer will display whatever you put between the parentheses of the print function, such as a word, a number, or a sentence in the console. We will use the print function very often in our programs whenever we want the computer to communicate something to the user. Below is our Hello World program, which prints "Hello, world!" to the console whenever it's executed. Run the program to see print in action!

hello.py

```
def main():
  print("Hello, world!")
if __name__ == '__main__':
  main()
```

Did you notice how, even though **"Hello, world!"** has quotations around it in the program, we don't see the quotation marks in the console. Python doesn't always know the difference between words and code, so we put quotation marks around words and phrases so that the program knows exactly what we mean.

Basics of Function Anatomy

print takes whatever you put inside its parentheses and prints it to the console. This might be confusing because, until now, you've just seen closed parentheses at the end of a function like move() or front_is_clear(). Many times, a function will need a bit of extra information before it can do its job. **print**, for example, doesn't know what you want it to write unless you tell it! The text you give **print** is called an **argument**, and we put the arguments of a function inside those parentheses at the end. Functions like move() don't have any arguments. They can do their job without any extra info from the programmer, so we just leave the parentheses empty.

Printing on a new line

After printing something to the console, Python moves down to the next line so that any additional text is separated from what you just printed. We see this come up when using multiple **print** statements in the same sentence:

hello.py

```python
def main():
   print("hello, world!")
   print("hello, code in place!")
   print("hello, friends!")
if __name__ == '__main__':
   main()
```

What can you print?

print works for many different types of text, including words, numbers, and symbols. If you want to print a number, just put the number in as an argument. If you want to put a word or a sentence, you need to surround it in quotation marks to tell the computer that it is looking at written words instead of Python code. **print** doesn't actually write out the quotation marks, but you have to include them as a programmer so that Python understands what you mean.

test_print.py

```python
def main():
   # Output: 2
   print(2)
   # Output: 3.14
   print(3.14)

   # Output: Hello
   print("Hello")
   # Output: Code in place!
   print("Code in place!")
   # Output: I luv 2 code!
   print("I luv 2 code")
   # Notice that even though 2 is a number, it still
   # goes inside the quotation marks!
if __name__ == '__main__':
   main()
```

"" vs ''

In other programming languages, there is a difference between single and double quotation marks. However, in Python, they are mostly equivalent. You can enclose words and sentences with *either* single or double quotes, but *not both!* 🛑

hello.py

```
def main():
    print("Hello, world!") # Perfect
    print('Hello, world!') # Yep!
    print("Hello, world!') # Very Bad!!!
if __name__ == '__main__':
    main()
```

We can be strategic about which set of quotes we want to use.

If your text contains single quotes, you should use double quotes:

print("no, you didn't") ->   **no, you didn't**

If your text contains double quotes, you should use single quotes:

print('say "hi" Karel') ->   **say "hi" Karel**

Advanced (Optional) Reading

There is actually a third type of quotation mark, and you've already seen it before! Remember block comments? Well, more generally, a block comment is just a sentence or paragraph that fits on multiple lines. We use triple quotation marks for these multi-line paragraphs, and we could actually tell the computer to print one out:

triple_quotes.py

```
def main():
    """
    You probably won't need to use triple quotation marks
    very often, but it is an extra tool you have available
    to you if you need it.

    Now, we're going to print this
    whole paragraph to the console!
    """
    print("""
    You probably won't need to use triple quotation marks
    very often, but it is an extra tool you have available
    to you if you need it.
    Now, we're going to print this
    whole paragraph to the console!
    """)
```

Separating Multiple Arguments

print can take in more than one value and print all of them together on a single line. We separate each value with a comma, and Python handles all the spacing for us. Multiple values do not have to be of the same type, and you can mix and match however you like:

multiple_arguments.py

```
def main():
  # Output: My name is Karel!
  print("My name is", "Karel!")
  # Output: "My name is Claire and I have 15 friends."
  print("My name is", "Claire", "and I have", 15, "friends.")
if __name__ == '__main__':
  main()
```

Escape Characters

When you press the return key to move to a new line, do you know how your computer keeps track of where the new line should begin? **Escape characters** are special symbols you can add between your quotation marks to stylize the text in various ways. Each escape character starts with a backward slash \ followed by a special code. There are several escape characters you can use, but below are the most common ones:

Code	**Meaning**	**Example**	**Result**
\’	Single Quote	‘I\’m a programmer!’	I’m a programmer
\n	New Line	“Good\nMorning”	Good Morning
\t	Tab	“Code\tIn Place”	Code In Place
\b	Backspace	‘Pytho\bn’	Pythn

Variables

Quest

You're well on your way to start writing your very own Python programs! With Karel, we worked with several different functions, the instructions that tell Karel how to interact with her environment. In Python, something else we can do is ask our computer to store information for us! After all, we humans can only remember so much at one time, and it's incredibly useful to let our computers handle all that information for us.

Variables

What are some examples of information we might want to keep track of?

- A person's name?
- How old is my friend?
- How many apples are left in my apple tree?
- Do I own a hat?
- What is the third letter of the English alphabet?

These are all things we could keep track of! As you can see, there are many different types of information you might want to store. Sometimes you are keeping track of a word, sometimes a number, or sometimes just a simple "Yes" or "No" like with "Do I own a hat?" As you might have guessed, your computer can store this information too.

When you want your computer to keep track of some data, you'll need to create a variable. Variables are made up of three things:

1. The value of your variable is the thing you want your computer to remember. A value could be something like a number or a word.
2. The name of the variable helps you keep track of what the value represents, and it's a description of the information stored in the variable.
3. The type of a variable is what sort of data we are putting inside our suitcase. Maybe we want to store someone's age, which is a number. What about their name, a word? Depending on what sort of information the computer is keeping track of, it will use a variable of a specific type.

Assigning a Variable

So how do we make one of these handy variables? We use something called the assignment operator! It's just the equals sign '='. However, it is important to remember when programming in Python that the equals sign *does not mean equals*!

Take this line of code, for example:

```
age = 22
```

We aren't just equating the word age with the number 22. Instead, we are telling the computer to assign the integer 22 to the variable called age.

The template for creating a new variable is as follows:

```
name = value
```

where *name* is the name of your variable and *value* is the data you want to be stored.

Memory Addresses

Your computer has tons of these things called memory addresses. You can think of them as little suitcases where you can store a piece of data. When we make a new variable, the computer finds an empty memory address and stores the value you give it inside that little suitcase.

To keep track of where it put your value, the computer assigns the location of that memory address to the name you chose for your variable. Think of this as a sort of luggage tag. Now, if you ever need to get that value back, you just use the name age, and the computer knows you mean, "I want the value assigned to the name age." The computer will then give you access to whatever is inside of the suitcase.

Here's how not to create a variable 🛑

bad_vars.py

```
def main():
   # Example 1
   22 = age
   # Not quite! When assigning to a variable, always put the name on the left side
       and the
   # value on the right
   # Example 2
   age
   # This is how you access the value of age. Unless 'age' is already a variable,
       this won't
   # make sense to your computer.
if __name__ == '__main__':
   main()
```

Let’s check out a program that uses one of these handy new variables:

```
def main():
   message = "Good Morning!"
   message = "Good Night!"
   print(message)
if __name__ == '__main__':
   main()
```

Can you guess just by reading this code what the program will print out? Let’s jump right in and see what’s going on.

```
message = "Good Morning"
```

In this first line here, we are creating a new variable called message and giving it a value of the string "Good Morning". We know we are creating a new variable because the name message hasn’t been used yet.

```
message = "Good Night!"
```

Here, we see message again, but we’re not actually assigning a new variable this time around. Once a variable is made with a certain name, using that name again now refers back to the variable you already created. This line is actually assigning a new string "Good Night!" to the name message.

```
print(message)
```

This is the print statement for our program. This time, message is just hanging out by itself, and that is because we aren’t trying to change its value anymore. Instead, we just want to know the value stored in that variable. When you write the name of a variable as a standalone word in your code, what you’re saying is, “Hey, computer, when you see this name, go get the current value of that variable and switch it with the variable's name here."

So, if we were to run this program, here's what would happen:

1. We create a new variable message and assign it the value "Good Morning!"
2. We assign message a new value "Good Night!"
3. We replace the word message in the print statement with its current value "Good Night!"
4. We run the instruction print("Good Night!"), which prints the string "Good Night!" to the console.

Using a Variable

As another example of accessing the value of a variable, your friend, Evan, wants you to keep track of how many pets he owns. He tells you that he has 4 pets, so you add the following line to your program:

```
evans_pets = 4
```

Now, if you ever need to reference how many pets Evan has, you can just use the name num_evans_pets. What if your other friend, Anna, tells you she has just as many pets as Evan? Instead of asking Evan how many pets he has and storing that number for Anna, you can just reference the variable that stores how many pets Evan has:

```
num_evans_pets = 4
num_annas_pets = num_evans_pets    # num_annas_pets now equals 4
```

Every Variable Has A Type

As we mentioned before, a variable has a **type**. Computers store words and sentences differently than it stores numbers. We call these different types **primitives**. The most common primitive types are shown below:

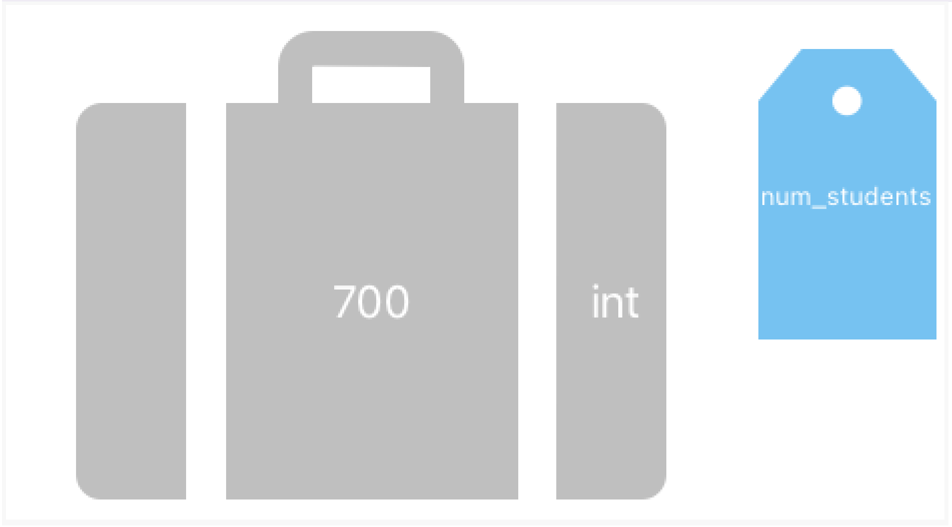
Type	Stores	Example
int (integer)	Whole numbers	137
float (floating point)	Decimal numbers	137.0
str (string)	text/character sequence between "" or ''	"Hello, World" *Note: The quotes won't appear when printing a string*
bool (boolean)	True or False	**True**

One thing to note about Python is that the type of a variable is not set in stone. You can assign a value of a different type to change the type of the variable:

```
def main():
  x = "hi"    # x is a string
  x = 7       # x is now an int
  print(x)
if __name__ == '__main__':
  main()
```

Even as you change variables, python keeps track of what types your variables are:

```
num_students = 700
```



Example: Python knows that num_students is an int value.

If num_students were to be reassigned to be a string, Python would update its type as well.

Swapping Two Variables

Oftentimes in coding, you will find that you have two variables, and you want to swap their values. On your first try, it might seem simple to just do something like this:

```
a = "Chris Piech"
b = "Mehran Sahami"

a = b              # a now points to "Mehran Sahami"
b = a              # b still points to "Mehran Sahami"
```

The code on a = b puts the **a** luggage tag on “Mehran Sahami” luggage. At this point, there is no luggage tag on the “Chris Piech” luggage so the luggage is lost. Both variables **a** and **b** are on the “Mehran Sahami” luggage and we have an unsuccessful swap :(

To fix this issue, we use a **temporary variable**.

```
a = "Chris Piech"
b = "Mehran Sahami"

temp = a                    # temp points to "Chris Piech"
a = b                 # a now points to "Mehran Sahami"
b = temp                    # b now points to "Chris Piech". We swapped!
```

The **temp** variable stores “Chris Piech” so that it is not lost when we reassign **a**. We have a successful swap! :)

Printing Variables

Sometimes, in order to debug our code, we will print out the value of a variable.

var.py

```
def main():
    name = "Karel"
    # Output: "Karel"
    print(name)
if __name__ == "__main__":
    main()
```

See how, although the word name is inside the print statement, the console actually prints the value of the variable called name, which is “Karel”

Forgetting to put quotation marks around your string literals is a very common bug ⚠️. If you forget your quotes, Python will think you are referring to a variable:

var.py

```
def main():
    karel = 7
    # Output: "Karel"
    print("karel")
    # Output: 7
    print(karel)
if __name__ == "__main__":
    main()
```

Notice how the one with quotation marks prints “karel” but the one without prints 7.

REPL

Quest

Ok, so now that we've learned about the console, variables, printing and IDE's, we can talk about another way to evaluate lines of code called the REPL

REPL stands for Read-Eval-Print Loop. It exists on most computer systems in the form of some sort of application that lets you execute code or commands (terminal in Mac, Windows terminal or command shell in Windows).

To use REPL in the terminal, you can simply type python:

```
$ python
>>>
```

The triple arrow shows up to let you know that you have entered the REPL. You can type any line of python code and hit enter and then the terminal will read the line, evaluate it, and print the results out to the user just like we would see in the Python Console!

The REPL is a great way to evaluate quick lines of code or even test small bits of code before putting them into a program.

```
$ python
>>> x = "the REPL is kind of cool"
>>> x
the REPL is kind of cool
```

One important thing to note: the REPL evaluates python code just like a normal interpreter. If the code contains an error 🛑 the error message will show up in the REPL just like it would in the python console.

```
$ python
>>> x =
File "<stdin>", line 1
    x =
      ^
SyntaxError: invalid syntax
```

The only case where this is not true is when the REPL thinks that you simply have yet to finish a line. For example, for unclosed parentheses, sometimes the REPL will simply output three dots … to show you that you have yet finish the statement:

```
$ python
>>> print("this line will not error"
...
...)
this line will not error
```

Each time you press return, you will be met with … until you close the parentheses. Then, the line will execute.

You can exit the REPL by pressing Ctrl+D and you should see your normal starting terminal line:

```
$ python
>>> [Ctrl+D]
$
```

So now you know how to execute Python code using two different systems, the console and the REPL! The console is for longer programs and programs that you want to save for later. The REPL is great for quick lines of code and testing the outcome of really short programs. To see some more examples of the cool things that you can do in the REPL, head to the next section on basic arithmetic!

Basic Arithmetic

Quest

Ok, for the last five sections, we have been in Computer Science class getting well acquainted with coding in Python. For this section, we are going to take a brief detour to Math class (but still in Python) to talk about how to do some basic mathematical operations in Python. By the end of this section, you should be able to do the four most simple operations—addition, subtraction, multiplication, and division—and understand the order of those operations. You should also understand the difference between an integer and a float, how to convert between the two, and how to account for conversion loss.

Int and Float

Before we dive into these concepts, let's begin with a brief reminder from the variables section. If you remember, in that section, we briefly defined int and float as follows:

An int is a number without a decimal point or a whole integer number.

A float is a number with a decimal point (including numbers with 0 after the decimal point such as 2.0). This will be useful to remember for the following concepts.

Python Arithmetic

The table below shows how to do some of the most basic operations in Python using examples from the REPL:

Operation	Symbol	Example

Addition	+	`$ python`
		`>>> 20 + 11`
		`31`

Subtraction	-	`>>> 17 - 23`
		`-6`

Multiplication	*	`>>> 15 * 12`
		`180`

Division	/	`>>> 10 / 2`
		`5.0`

An important thing to note: dividing by zero will result in a ZeroDivisionError 🛑:

```
$ python
>>> 6 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Type Conversion

Aside from errors, one thing that you may have noticed in the table is that using the / symbol will result in a float result (even if both of the arguments are integers). This is called implicit type conversion. This shows up a lot in python. That was an operation between two integers, but any operation between an int and a float will result in a float:

conversion.py

```
def main():
  x = 80 + 16.0
  y = 30.0 - 37
  z = 41 * 82.0
  print(x)
  print(y)
  print(z)
if __name__ == '__main__':
  main()
```

Fortunately, in Python we do not need to worry about the distinction between floats and ints too much, but it is a good thing to know if you are wanting whole numbers and getting decimals. A distinction that we do need to worry about however is strings vs ints and floats. If you use the plus sign operator **'+'** between two strings, even if the two strings are of numbers (ex. '15'), Python does not treat them like ints or floats. When using + between two strings, Python **concatenates** them. Consider the example below:

conversion.py

```
def main():
  x = '80' + '16.0'
  print(x)
if __name__ == '__main__':
  main()
```

Instead of adding the two numbers, Python essentially glues the two strings together to form one string. This works with non numbers too.

hello.py

```
def main():
  x = 'Hello'
  y = 'Code in Place!'
  print(x + ' ' + y)
if __name__ == '__main__':
  main()
```

Notice how we have to add a space in between the two strings. Otherwise we would have gotten 'HelloCode in Place' which isn't quite right. If you read the optional section in Print on printing with multiple arguments, you might note that this is slightly different from the **print** function which automatically adds a space between the arguments. Another important thing to note is that python does not have implicit type conversion between strings and numbers. Trying to add or concatenate a string and an int or float will lead to an error 🛑:

conversion.py

```
def main():
  x = '80' + 16.0
  print(x)
if __name__ == '__main__':
  main()
```

Thankfully, there is also **explicit type conversion**. This is accomplished without any operations but rather by simply calling the int() and float() functions.

conversion.py

```
def main():
  x = int('34') + 17.0
  y = 19 + int(float('5.0'))
  print(x)
  print(y)
if __name__ == '__main__':
  main()
```

The conversion functions only do one conversion at a time. You can't call the int() function on a string that has a decimal point. You have to convert the string to a float first and then an int.

Also, it is important to note that the int() function simply chops off anything after the decimal. No rounding is involved. This can be dangerous. If we convert a float to an int and then back again, we might lose information. This is called **conversion loss.**

conversion.py

```
def main():
  x = 3.625
  y = int(x)
  print(float(y))
if __name__ == '__main__':
  main()
```

As we can see when we run it, the .625 is lost as a result of calling the int() function.

Order of Operations (so far)

The precedence of operators or order of operations up until this point is as follows:

highest

() parentheses
* /
+ -

lowest

We will add to this list of precedence in the conditionals and advanced arithmetic sections. For now let's focus on the three that we have filled. As you can see in the figure above, parentheses are evaluated first, followed by multiplication and division and then addition and subtraction. When two operations have the same precedence and are on the same line (and not separated by parentheses) the order of precedence is left to right with the leftmost operator being evaluated first:

```
$ python
>>> 30 / 3 * 5
50.0
```

In this way, Python follows the typical order of operations found in regular math.

Now let's see some more examples. Swipe through these slides to see some examples of how expressions are evaluated and maybe try to answer a few on your own before clicking to the answer slide:

Assigning Expressions to Variables

In the variables section, we showed you how to store things like **ints** and **floats** in a variable using the assignment operator with a simple statement like x = 1068. The cool thing about Python is that we can also store expressions in variables. This is particularly useful when doing operations between two variables.

diff.py

```
def main():
    x = 99
    y = 70
    z = x - y
    print(z)
if __name__ == '__main__':
    main()
```

In this line z = x - y, **z** is being set equal to the **values** represented by x and y. It is essentially receiving a copy. Therefore, if we update **x** or **y**, the value stored in **z** does not change:

diff.py

```
def main():
    x = 99
    y = 70
    z = x - y
    x = 37
    print(z)
if __name__ == '__main__':
    main()
```

We can even do this without upsetting the program:

diff.py

```
def main():
    x = 99
    y = 70
    z = x - y
    x = z - x
    print(x)
if __name__ == '__main__':
    main()
```

In the above example, z is set equal to x - y or **99** - **70** which equals **29**. Then x is updated to equal z - x or **29** - **99** which equals **-70**. z is unaffected by that line because the assignment to z is not a formula that Python has to follow but a one time assignment of the value. Another way to put it is that the expression on the right hand side of the equals sign is evaluated first and then assigned to z. The variable never knows what operations it took to get to the value it is assigned. It only knows its value.

Operation Shortcuts

As you have seen in many examples in this section, we can update a variable’s value by using that same variable’s name.

increment.py

```
def main():
   x = 7
   x = x + 1
   print(x)
if __name__ == '__main__':
   main()
```

In fact, there are many cases in which we want to do something like this (such as incrementing a counter). Because of this, Python has a built in shortcut to write such a statement. If we want to say `x = x + 3` we can simply write `x += 3`. This is true for all of the operations we have learned about in this section. As a general rule:

variable = variable operator (expression)

is equivalent to

variable operator= (expression)

Here are more examples below:

shortcut.py

```
def main():
  x = 15
  x += 6   # same as x = x + 6     (x = 21)
  x -= 10  # same as x = x - 10    (x = 11)
  x *= 8   # same as x = x * 8     (x = 88)
  x /= 4   # same as x = x / 4
  print(x)
if __name__ == '__main__':
   main()
```

In order to do this, **x** has to have already been assigned a value. If we try to do these statements (even the longhand version) before x has been assigned a value, we will get an error 🛑

```
$ python
>>> x += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Practice

If you want to get some more practice with these concepts, flip through these slides and determine what each expression will evaluate to in Python. Then check your answer on the following slide.

User Input

Quest

Wow! You have mastered the world of variables and basic arithmetic in Python! You can now store information in variables and do operations with those variables if they represent numbers. This is great for when you have data that you want the program to remember, but what if you want to store input from the user?

By the end of this section, you should be able to get input from the user and understand how to properly access it. So what is the secret behind getting user input? Drumroll please….

🥁🥁🥁🥁🥁🥁🥁🥁🥁🥁🥁🥁🥁

the input function!

The syntax of the input function is written below:

```
answer = input('message/prompt to the user')
```

The message inside the parentheses will be outputted to the console just like the inside of a print statement. Then the console will wait for the user to respond and hit enter. Whatever the user responds with will be stored in the variable answer.

Here’s another example of what this might look like running in the console:

hello.py

```
def main():
   print('hello there')
   user_status = input('how are you? ')
   print('you said: ' + user_status)
if __name__ == "__main__":
   main()
```

Note: running this in the reader should create an input text box, but the prompt may not be there. Just type in your response and hit enter to see the result.

An important thing to note is that the value stored from a call to input will always be a **string** type. If you wish to get an **int** or **float** value, you have to use the built in type conversion functions that we talked about in the arithmetic section:

divide_by_two.py

```
def main():
   x = input('enter a value:')
   print(x)
```

```
    print(float(x) / 2)
if __name__ == "__main__":
    main()
```

Note: running this in the reader should create an input text box, but the prompt may not be there. Just type in a numerical value and hit enter to see the result.

A question that you might be having is what happens if the user doesn’t input anything? If the user just hits enter, then the value stored in the variable will just be nothing.

value_input.py

```
def main():
    x = input('enter a value: ')
    print(x)
if __name__ == "__main__":
    main()
```

Note: running this in the reader should create an input text box, but the prompt may not be there. Just hit enter to see what happens when the user doesn't input anything.

As you can see, the result is just a blank screen instead of an error.

Conditionals

Quest

True or false, is coding the coolest thing ever? We sure think so, but you could've said false. We use boolean variables to store True or False values. You've seen functions like Karel's front_is_clear() that return boolean values. Ever wonder how these functions actually work? Well, under the hood, boolean functions often rely on asking a computer questions about the data it is storing, then returning True or False based on the answer. In this section, we will see some of the fundamental questions you can ask about data, and how we can build upon these questions to make powerful programs.

Comparison Operations

Which is heavier, a pound of rocks or a pound of feathers? Actually, it's a trick question, because *both* weigh exactly one pound! The idea of comparing two things is something that will come up *a ton* when programming. Luckily for us, Python has just what we need to get the job done! Each comparison operator takes two values and reasons something about their relationship to one another. The result of the operation is a boolean value indicating whether the condition of that particular operation was met. Here are the comparison operators, and an example of when each would return True and False:

Comparison Operator	Meaning	True Example	False Example
==	Equal	1+1 == 2	2 == 3
!=	Does not equal	3.2 != 3	5-5 != 0
<	Less than	5 < 7	2 < 1
>	Greater than	4 > 2	1 > 9
<=	Less than or equal to	90 <= 100	18.4 <= 4
>=	Greater than or equal to	5.0 >= 5.0	11 >= 20

Logical Operations

Sometimes, one comparison operation isn't enough. Say you are writing a program for an online bakery. The bakery limits each customer to a dozen cookies, and they want you to make sure nobody orders more cookies than what is left in stock. Each time a person places an order, you now have two things to check:

1. Did they order a dozen cookies or less?
2. Do we have enough cookies in stock to fill this order?

We could ask these questions one at a time, but wouldn't it be handy if there was a way to check if *both* were True at the same time? Spoiler alert, you totally can, through the magic of logical operations. Just like comparison operations, these logical operations will also return a boolean value. The difference is that comparison operations take in variables, while logical operations take in other boolean values. Here are the three main logical operators you will use in your programs:

Operator	True Example	False Example
and	(3 < 5) and (-1 == -1)	(7 == 8) and (12 > 2)
or	(0 == 1) or (10 <= 15)	(0 > 5) or (6 == 3)
not	not False	not 1 > 0

The and operator checks if two statements are both True, while the or operator checks if one or both are True. It is important to realize that this is an *inclusive or*, meaning all we care about is that *at least* one of the two is True. The not operator is interesting. It returns True if the statement on the inside is False. You might look at not(2==3) and say "Wait a minute, couldn't you just say *2 != 3.*" Both would do the exact same time. The not operator might seem redundant in the context of comparison operations, but it will become much more useful in the next section about expressions, so read on if you want to learn more!

Logical Expressions

A logical expression is one or more boolean values connected with logical operations. These boolean values could come from anywhere, a comparison operation, a logical operation, a boolean function, boolean variable, or even a literal boolean value True or False. Just like before, the entire logical expression will evaluate True or False, so we can use it as a conditional just like we would any other boolean:

```
def main():
    name = "Michael"
    age = 21
    born_in_december = True
    lucky_number = 6
    if name != "Karel":
        print("Hi, I don't think we've met before!")
```

```
   if age == 21 or not(born_in_december):
      print("Do you remember...")
   if lucky_number > 5 and name == "Michael" and not(age == 30) and
        born_in_december:
      print("Hey! I've been looking all over for you!")
if __name__ == '__main__':
   main()
```

All three of these expressions are true. Can you see why?

Precedence

In Python's logical order of operations, comparison operations are evaluated before logical operations. An operator has greater **precedence** if Python evaluates it before another. If two operations have the same precedence, then Python evaluates whichever comes first in the expression. Even though the third logical expression above contained many comparisons and logical operators, Python just reads them left to right and had no trouble understanding what we meant. However, once you start mixing and matching **and**, **or**, and **not** together, things can get messy really fast. Consider the following logical expression:

```
a or not(b and c) and d or e and f or not(g)
```

This is a nightmare!!! Python will have no trouble understanding this because of its order of operations, but writing expressions in this way is destined to lead to mistakes. If you're not careful, an expression like this could end up meaning something very different than what you intended. However, check out Python's order of operations so far…

highest

() parentheses
* /
+ -
comparison
not
and or

lowest

No matter what, parentheses always go first! We can use this to our advantage. By grouping related operations together with parentheses, we can make our logical expressions much more reader to understand what we are coding and to avoid sneaky bugs:

```
# Before
a or not(b and c) and d or e and f or not(g)

# After
(a or not(b and c)) and (d or e) and (f or not(g))
```

Not only do the parentheses make this expression easier to read, where we put them affects what the expression means logically. Rather than going back and adding parentheses after you write your expression, it's a good habit to use them from the start. This will also just make writing expressions a lot easier.

If you're up for a fun challenge, which of the following expressions do you think will evaluate to **True**? Don't run it until you have your answer!

```
def main():
  a = True
  b = False
  c = False
  d = True

  # Option 1
  if (a and not(b)) and (b or (d and a)):
         print("Option 1 is True!")

  # Option 2
  if a and (b or (d and not(not(c) or (a and d and c)))):
         print("Option 2 is True!")

  # Option 3
  if (a and not(b) and not(c) and d and (a and d) and not(b or c)) and not(d and (a or
(b and c)) and c):
    print("Option 3 is True!")

if __name__ == "__main__":
       main()
```

Loops

Quest

You’ve been working so hard, and here you get to rest and stretch your legs. We’re just going to review loops like you’ve already seen in Karel already, as they work exactly the same way in Python. We’ll also throw in a few extra things that are good to know, but this section will feel like a nice review.

For vs While Loops

A for-loop repeats a block of code a fixed number of times, incrementing a counter at each repetition. A while-loop repeats a block code an indefinite amount of time until some condition is no longer met. The rule of thumb for choosing which loop to use is as follows:

- *If you know how many times your loop will run, use a for-loop.*
- *If you don’t know how many times your loop will run, use a while-loop.*

For Loops

A for-loop as three parts:

1. Loop Variable - the variable that keeps track of count
2. Range - which numbers should we loop over
3. Code Body - the block of code that gets repeated at each iteration

The most basic for-loop is one that loops for all values from 0 to some number. We use the range function to specify which number that is:

```
def main():
  for i in range(5):
      print("I can count to", i + 1)
if __name__ == "__main__":
   main()
```

range()

Most of the time, you’ll just use the range function to set the number of times you want your code to loop. range is more powerful than this, though. What range really does is present the for-loop with all the values that i should take on, and the for-loop just repeats once for each of those values. When we say

```
for i in range(some number):
```

we really mean “repeat the following code one time for each value of i from 0 to *some number*”.

Your code might need to be more specific than this. Maybe you want to start somewhere besides 0, and instead only loop over all values from 5 to 10. Maybe you only want to loop over even numbers in a range. range() can do all of these things!

Here is a brief guide for using the range function in various ways:

All numbers **0** to stop (**stop** is not included)	All numbers **start** to **stop** (**stop** is not included)	All numbers **start** to **stop**, taking **step** steps at a time
range(stop)	range(start, stop)	range(start, stop, step)
Ex: range(10) gives you 0,1,2,3,4,5,6,7,8,9	**Ex:** range(2,8) gives you 2, 3, 4, 5, 6, 7	**Ex:** range(0,10,2) gives you 0, 2, 4, 6, 8

While Loops

while-loops are *much* simpler than for-loops! They just run until some condition returns false:

```
def main():
  print("Pop Quiz! How many continents are there on Earth?")
  answer = int(input())
  while(answer != 7):
    print("Not quite, guess again!")
    answer = int(input())
  print("That's right! There are seven continents on Earth!")
if __name__ == '__main__':
  main()
```

In the above program, the while-loop checks each time it repeats whether the user has inputted the correct answer. If the current answer is not 7, the expression answer != 7 evaluates to **True** and the loop repeats once more. Eventually, if the user answers 7, answer != 7 evaluates to **False** and the loop ends.

How many times do we need to repeat before the user gets the question right? We don't actually know! They could get it on their first try or they could never get it right. **while-loops** are perfect for this sort of situation. With a **for-loop**, we would have to set some maximum number of attempts before the loop forcibly ended, whereas a while-loop can go on indefinitely until the user gets the right answer.

Infinite Loops

It is important that we don't actually write code that will never stop. An **infinite loop** is when your code gets stuck in an endless cycle of repeating code and never finishes. It's pretty hard to get a Python for-loop stuck in an infinite loop, but we need to be especially careful with while-loops. Let’s look at a couple of examples of code that will run forever:

```
# The key to thwarting an infinite loop is to loop for conditions that will never be False

# The boolean value True will never be False
while True:
    # Do something

# i keeps decreasing and will always be less than 1
i = 0
while i < 1:
    i = i - 1

# Since two values are always either equal or not equal this
# condition will never be false
a = True
b = False
while (a == b) or (a != b):
    # Do something
```

Nested Loops

Quest

Many programs use loops to repeat an instruction multiple times, but some programs need to repeat the loop itself – they loop over the loop! Amazing things can happen when we realize that a loop repeats *any* block of code, including another loop! Sometimes by simply using the tools you already have in a new way, you can unlock a whole new skill you didn't know you had! If you can believe it, the next section won't cover any new keywords, and yet you will still walk away with a new programming skill. Don't believe me? Let's see below!

Nested For Loops

What do you think this code does?

```
def main():
  for i in range(5):
    for j in range(10):
      print("Hello, World!")
if __name__ == "__main__":
  main()
```

A for-loop *inside* a for-loop? How strange! But if we read this code carefully we should be able to figure out what it does…

The first loop is going to repeat whatever is inside 5 times. However, the stuff inside is another loop! This inner loop prints **"Hello, World!"** 10 times. So, translating this into words, the function goes something like this:

- Do the following 5 times
  - Print **"Hello, World!"** 10 times

You have just discovered **nested loops**. We called them *nested* loops because one is inside the other. Nested loops are great if you have a task you want to repeat but you also want to do that repetition multiple times. For now, we are going to keep the examples pretty simple to get you familiar with using nested loops, but you'll come to see that this way of programming becomes much more useful when we talk about more advanced topics like graphics. Stay tuned!

Nesting For and While Loops:

Like we said, loops don't really care what they are repeating. You are free to nest for-loops inside while-loops and vise-versa. Also, very important, the inner loop doesn't have to be the only thing inside the outer loop.

```
def main():
   number = 64
   while (number >= 1):
      print("I'm going to count to three!")
      for i in range(3):
         # Remember range() starts at 0
         print(i+1)
      number /= 2
if __name__ == "__main__":
   main()
```

Did you notice the following line in the previous example:

```
print("I'm going to count to three!")
```

We placed this line *inside* the outer loop but *outside* the inner loop (Say *that* ten times fast!) When using a nested loop, you need to be careful where the repeated code is placed. This is because code within the inner loop runs more often than code that's just in the outer loop. What would happen if we moved that print statement inside the inner loop? Try moving that print statement inside the inner loop and see what happens!

Seriously… give it a go!

```
def main():
   number = 64
   while (number >= 1):
      for i in range(3):
         # Remember range() starts at 0
         print("I'm going to count to three!")
         print(i+1)
      number /= 2
if __name__ == "__main__":
   main()
```

Wow, that's way too many prints! The problem is that we only want this line to run when we begin a new count of three, but right now it is run every single time we count off a new number. What might help is to think of the inner loop like a function and ask yourself whether the code you are adding should be apart of that function:

```
def count_to_three():
   for i in range(3):
      # Remember range() starts at 0
    print(i+1)
def main():
   number = 64
```

```
  while (number >= 1):
    print("I'm going to count to three!")
    count_to_three()
    number /= 2
if __name__ == "__main__":
  main()
```

Thinking about it this way, can you see why the print statement should stay outside the inner loop?

Don't Reuse Loop Variables

We typically use the variable **i** as the loop variable for a for-loop. When nesting for-loops inside each other, we need to pick a new variable name for the loop variable of the inner loop. Typically, programmers use **j** because it comes right after **i** in the alphabet:

```
def main():
  for i in range(10):
    print("I'm going to count to three!")
    for j in range(3):
      print(j+1)
if __name__ == "__main__":
  main()
```

Reusing Outer Variables

We can make use of the outer loop’s loop variable when writing the code for our inner loop. Take a second to think about how this is useful. We can write code for our inner loop that changes depending on which iteration of the outer loop we are on. Try to guess what this code does before you run it!

```
def main():
  for i in range(10):
    for j in range(10):
      print(i)
if __name__ == "__main__":
  main()
```

Deeper Nesting

There’s nothing stopping you from nesting another loop inside that inner loop. You can nest as many loops as you like, but be warned! As you nest more and more loops inside of each other, your program is going to get really long really fast. Even a triple-nested loop, while very literally a short amount of code, could take a huge amount of time to finish running. By convention, the variable name for the iterator of each loop is the letter after the previous nested loop:

```
def main():
        # Times "Code in Place" is printed: 10
        for i in range(10):
                print("Code in Place")

        # Times "Code in Place" is printed: 100
        for i in range(10):
                for j in range(10):
                        print("Code in Place")

  # Times "Code in Place" is printed: 1000
        for i in range(10):
                for j in range(10):
                        for k in range(10):
                                print("Code in Place")


if __name__ == "__main__":
        main()
```

Random

Quest

Now that we have revisited if statements and loops and learned how to use each in Python, we are ready for some more advanced introductory topics! First up is a very useful Python library: random.

There are lots of situations where we might want to use random numbers. For example, many games use random numbers for things like dice rolls. To get access to the random library, we simply use the now familiar import keyword:

```
import random
```

By the end of this section, you should be able to generate both random ints and floats of varying ranges by using the random library's most basic functions.

Useful Functions

This will give us access to all of the functions in the random library. Here are a few of the most useful functions for this class:

Function	What it does
random.randint(min, max)	Returns a random int between min and max inclusive
random.random()	Returns a random float between 0 and 1
random.uniform(min, max)	Returns a random float between min and max
random.seed(1)	Fixes the random generator so that it gives the same sequence of numbers each time. Can change 1 to another number for a different fixed sequence. Helpful for debugging!

Let's walk through them one by one.

random.randint

Let's start with an example. Let's say that you are building a game, and you decide that you want to build in a roll dice function. Using the randint function, you might do something like this:

single_dice.py

```
import random
def main():
  # generate random number between min 1 and max 6
  dice_roll = random.randint(1, 6)
  print('dice value:', dice_roll)
if __name__ == "__main__":
  main()
```

If you run the above program a bunch of times, you won't always get the same result. Up until now, we've only seen programs that do the same thing. This opens up a whole new world of possibilities! Another thing to note is that both 1 and 6 are possible outputs of randint because the function is **inclusive**.

random.random

Now that we've seen randint, let's explore another way to generate random numbers. The random.random function gives back a float between 0.0 and 1.0 (values can include 0.0 but not 1.0) This can be useful when we want to get random numbers that have decimal values.

Let's apply this to our game. Now that we have dice rolling, we can focus on the game board. The board has squares and each square has the possibility of being a bomb square. If players land on a bomb square, they have to go back to the start. To determine whether a square is a bomb or not, we can assign it a random probability.

You do not need to understand probability for this class. Without going into too much detail, a probability is a value between 0 and 1 that describes the likelihood of something happening. In this case, the likelihood of being a bomb square.

If the square's probability is larger than a certain value (we'll call this the threshold) then we will determine it is a bomb.

square.py

```
import random
# generate random probability between 0 and 1 (including 0)
def main():
  threshold = 0.80
  # get the probability
  probability = random.random()
  # determine if the square is a bomb
  is_bomb_square = False
  if probability > threshold:
    is_bomb_square = True
  print(probability)
  print(is_bomb_square)
if __name__ == "__main__":
  main()
```

Once again, run this multiple times to see the different results.

random.uniform

Ok, so our game has dice. It has a game board. What about the avatars? Let's say that we want each player to have a unique avatar for playing this online board game. We want each avatar to be unique so that players have lots of options to choose from. To create some variety in the avatars, the first thing that we want to do is give each avatar a random height.

Our avatars aren't human, so their heights can range anywhere from 1.0 to 3.0 meters (3 feet 3 inches to 9 feet 10 inches for our American students).

This means that we want float values, so we can't use randint, but we also want our values to be bigger than 1, so we can't use random.random. To get our random heights, we will have to use random.uniform.

The numbers generated from random.uniform go out many many decimal places, but we only want to use go out to 3. We can use python's built in round function to round it to 3 decimal places. The round function takes in the number to round and the number of decimal places to round it to.

height.py

```
import random
def main():
  # generate random height between 1.0 and 3.0 (inclusive)
  height_in_meters = random.uniform(1.0, 3.0)
  height_in_meters = round(height_in_meters, 3)
  print("height: " + str(height_in_meters) + "m")
if __name__ == "__main__":
  main()
```

Run this multiple times to see the different results.

One thing to note: like random.random, random.uniform is **inclusive** which means that the values it outputs can include min and max.

random.seed

In order to make random numbers, Python’s random library uses a **random number generator**. You don’t need to understand how this generator works, but one thing that is useful to know is that it can be fixed.

seed.py

```
import random
def main():
  random.seed(1)
  print(random.randint(0, 100))
  random.seed(1)
  print(random.randint(0, 100))
  random.seed(0) # different seed gives different number
  print(random.randint(0, 100))
if __name__ == "__main__":
  main()
```

As you can see, `random.randint` gave the same number twice. Using the seed method ensures that the generator will give the same sequence of numbers every time, but only for the same seed. It is also important to note that you have to say `random.seed()` every time you want the sequence to be the same.

It might seem counterintuitive to have a function that makes random number generation less random, but this becomes useful when we are trying to debug a program involving random numbers (especially if a program seems to work for some random numbers and not others).

Nonetypes

Quest

Now, you've been introduced to the random library. You understand how to create random values for two different variable types: ints and floats. In this section, we are going to take a very brief look at a special variable type called None.

None is the keyword that Python uses to indicate when a variable has no value.

```
x = None
```

Why would we want a variable to have no value? Well, sometimes when writing a program, we don't know what we want the value of a variable to be at the time when the variable is created. The None value can serve as a sort of placeholder for when we need to create a variable before we know what value it needs to have.

Functions Can Also Give the Value None

The example above is not the only way to get a variable to equal None. If you recall from the User Input section, we have seen how variables can be set to equal the result of a function call. Well what happens if the function that we call does not return anything? Do we just get an error?

The answer is no. It is perfectly fine to set a variable equal to the result of a function that returns void. The result is that the variable will store the value None:

```
$ Python
>>> x = print("this function returns nothing")
this function returns nothing
>>> x
None
```

Checking for None

None is a cool concept to have, but it can cause errors in our code if we are not expecting it. For example, imagine that we have a program that uses a function called mystery_int(). mystery_int() returns an integer some of the time and None otherwise. If mystery_int() returns an int, then we want to print the square of it. Don't worry too much about the actual code for mystery_int(). We will cover the return keyword in the functions section later on.

Look at the code below to see how this might work:

mystery_squared.py

```
import random
def mystery_int():
   """
   return 2 or None with equal
   probability (0.5)
   """
   if random.random() > 0.5:
      return 2
def main():
  """
  set x equal to mystery_int
  print the square of x
  """
  x = mystery_int()
  print(x)
  print(x * x)
if __name__ == "__main__":
   main()
```

If you run the above code, there's about a 50% chance that it worked, and a 50% chance that you got an error. So if you got an error 🛑 what happened?! Well, mystery_int() gave us **None** instead of an integer. Because NoneType is not a number, we cannot use the * operator on it. The result is an error. So, how do we fix this? We can use if statements! NoneType is a comparable object which means that we can use the **==** and != comparison operators on it. If you check to see if any actual value == **None**, you will get **False**. Let's update our code.

mystery_squared.py

```
import random
def mystery_int():
   """
   return 2 or None with equal
   probability (0.5)
   """
   if random.random() > 0.5:
      return 2
def main():
  """
  set x equal to mystery_int
  check if x is an integer
  if so, print the square of x
  """
  x = mystery_int()
  print(x)
  if x is not None:
    print(x * x)
if __name__ == "__main__":
  main()
```

Yay! Now when mystery_int() gives us a value of **None**, we can still have a working program.

Advanced Arithmetic

Quest

Welcome to Math class part 2! In this section, we will go over some of the more advanced operators in Python (including integer division, exponentiation, modulus, and unary negation) and give you a brief introduction into the Python Math class.

Advanced Arithmetic

This table below has the last four math operators that you will need to complete your mathematical tool belt.

Operation	Symbol	Example

Integer Division	//	$ python
		>>> 10 // 4
		2

Exponentiation	**	>>> 2 ** 3
		8
		>>> 2 ** -1
		0.5

Modulus	%	>>> 8 % 3 2

Negation	-	>>> x = 0.5
		>>> -x
		-0.5

Integer Division

As we saw in the Basic Arithmetic section, using the division symbol / in Python always results in a float value. If we want to drop the decimal, we can simply use the integer division operator //. This operator is the equivalent of calculating the division the normal way and then rounding down.

int_div.py

```
def main():
    x = 36 / 10
    y = 36 // 10
    print(x)
    print(y)
if __name__ == '__main__':
    main()
```

This means that integer division can also result in **conversion loss**. In the example above the 0.6 is completely lost information in the variable **y**.

A few special cases to consider below:

int_div.py

```
def main():
    x = 36.0 // 10
    y = -5 // 2
    print(x)
    print(y)
if __name__ == '__main__':
    main()
```

Because of the implicit type conversion that we discussed in the last section, using the // operator with a float will result in the float equivalent of the rounded number. This can be confusing because this is called **integer division** but as a rule of thumb, **any** operation between an int and a float will result in a float.

The second example is just meant to show what rounding down looks like for negative numbers. -5 / 2 is -2.5 and because integer division rounds down instead of truncating the value, the result is -3 (since -3 is less than -2).

Exponentiation

As we can see in the table, the symbol for exponents is ** and it supports negative exponents. The important thing to note is that negative exponents will give **float** values even if both arguments are ints. This is another example we've seen of **implicit** type conversion between two ints.

Modulus

The modulus operator gives you the remainder of a division between two numbers.

mod.py

```
def main():
    x = 39 % 17
    y = 12 % 4
    z = 18 % 19
    print(x)
    print(y)
    print(z)
if __name__ == '__main__':
    main()
```

If the second number divides evenly into the first, the result will be 0. If the second number is larger than the first, the result will be the first number.

Use the modulus operator when both numbers are positive and ints. Don't worry about negative numbers or floats for now.

Similar to dividing by 0, modulus or integer division by zero will result in a ZeroDivisionError 🛑:

```
$ python
>>> 7 % 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 32 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Unary Negation

As you may have noticed, the symbol for unary negation and the symbol for subtraction are the same. This symbol is called **unary** because it only acts on one thing. The expression to the right of the operator is negated (if it is positive, it becomes negative and vice versa).

negation.py

```
def main():
    x = - (19 - 7) # the 1st is minus, the 2nd is negation
    y = 23 - -4    # the 1st is negation, the 2nd is minus
    z = --8        # both negation
    print(x)
    print(y)
    print(z)
if __name__ == '__main__':
    main()
```

As you can see in the last example, two unary negation signs cancel each other out.

Precedence

The figure below shows the final order of precedence table.

highest

() parentheses
** exponentiation
- negation
* / // %
+ -
comparison
not
and or

lowest

As we mentioned before, operators of the same precedence are evaluated in order from left to right. Click through the slides to see some examples, and as always, try to figure out the answer before you click on the answer slide.

Math Library

In addition to operators, Python also has a Math library to help with mathematical operations. First, we must import the library as usual:

```
import math
```

The **Math** library has two important things that we are going to talk about: constants and functions. Some useful constants are listed below:

Value	**Python Representation**
π	math.pi
e	math.e

Next up is useful functions:

What It Does	**Function**
x	math.sqrt(x)
e^x	math.exp(x)
ln(x)	math.log(x)

To see an example, let’s say you wanted to represent euler’s formula in python:

$$e^{i\pi} + 1 = 0$$

Note: $i = \sqrt{-1}$

euler.py

```
import math
import random       # to give us random values for x and y
def main():
   x = random.randint(0, 10)       # must be a non-negative number
   y = random.randint(1, 10)       # must be a positive number
   ans = math.exp(math.sqrt(x) * math.pi)
   ans += math.log(y)
   print(ans)
if __name__ == '__main__':
   main()
```

Uh oh! We’ve reached an error 🛑. Negative numbers are outside of the domain of square roots, so when we try to call math.sqrt(-1) we get an error. The same is true for calling math.log() on zero or negative numbers. Domain rules apply the same in Python as they do in regular math. So, Euler’s formula is not going to work using what we’ve learned so far. Instead, let’s make a new formula. We’ll call it the Code in Place formula:

$$e^{\sqrt{x}*\Pi} + ln(y)$$

Let’s see what this evaluates to using Python!

CIP_formula.py

```
import math
import random       # to give us random values for x and y
def main():
   x = random.randint(0, 10)       # must be a non-negative number
   y = random.randint(1, 10)       # must be a positive number
   ans = math.exp(math.sqrt(x) * math.pi)
   ans += math.log(y)
   print(ans)
if __name__ == '__main__':
```

```
main()
```

Ok, that’s an interesting number, but more importantly, we can now combine our skills of operators and the math class in python! Yay!!

Practice

Just like in the last arithmetic section, if you would like some extra practice, flip through these slides and try to evaluate each line before clicking to the answer slide.

Floating Point

Quest

This section will cover a picky detail about how computers store numerical values. While you might not encounter this specific problem very often, it's important to be aware of it when writing your program to avoid bugs. You know that Python has integers, which are whole numbers, and floats, which are decimals. These floating-point numbers as they are officially called get their name from the way the decimal point can *float* to different positions in the number, changing the value that is represented. Float values can be written as a regular decimal, like 3.14, or in scientific notation with the letter ‘e’ or ‘E’, like this:

1.2e23 ⇒ 1.2 * 10^23

1.0e-4 ⇒ 1.0 * 10^-4

Your computer only has so much memory. If you have a number like ⅓ , storing that as a float would require an infinite amount of memory (0.333333333333…). This means we can only store the value as an approximation. We talk about the precision of a floating-point number as how many decimal places of the true value are accurately recorded.

0.3 is an approximation of ⅓

0.33 is a more precise approximation

0.333 is an even more precise approximation

Your computer can store decimal values very precisely, up to 17 decimal places of precision, but this means there are some values too small to be stored in a float (think of something like 1.0E-1000000000).

When programming, you might use a math expression to calculate a decimal value. However, if the result of that expression is a value too small to be stored accurately, Python will truncate the tail end of the value, resulting in a loss of precision. This can lead to all sorts of weird quirks when doing arithmetic:

```
$ python
>>> 0.1 * 3
0.30000000000000004
```

Check out what happens when we run this program. The code continuously divides a value by 10 until eventually Python can’t store so much precision and just rounds it down.

floating_point.py

```
def main():
  value = 1
  while value > 0:
    print(value)
    value /= 10
  print(value)
if __name__ == '__main__':
   main()
```

When Python truncates the end of a floating-point number, the result is a value incredibly close to but not quite exactly the desired value. The difference between the approximation is small, sure, but it could still be disastrous for your code. Below is an example of common errors related to floating-point precision:

floating_point2.py

```
def main():
    # The actual value stored in this variable is 0
    denominator = 1E-1000000

    # ZeroDivisionError (You can't divide by 0)
    value = 10 / denominator

    if denominator == 0:
        # This code should never run but it does
        print("Something has gone horribly wrong :(")

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

— Graphics —

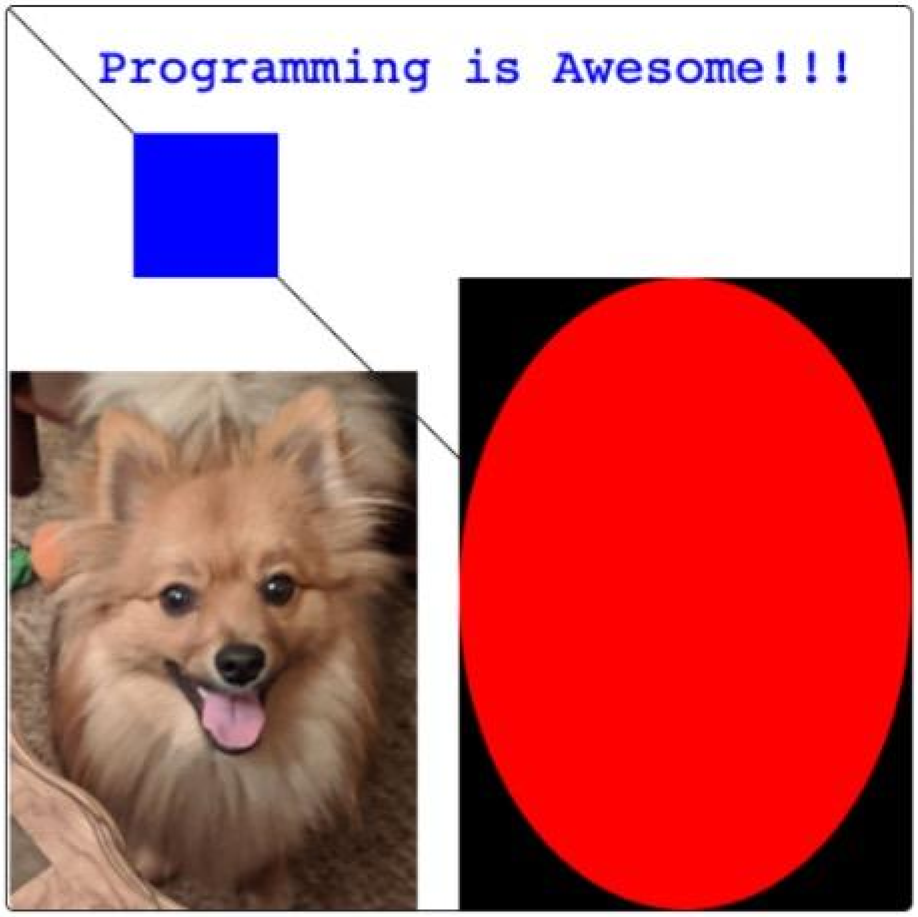
Basic Shapes

Introduction to Graphics

Adapted from the Stanford CS106A Graphics Reference

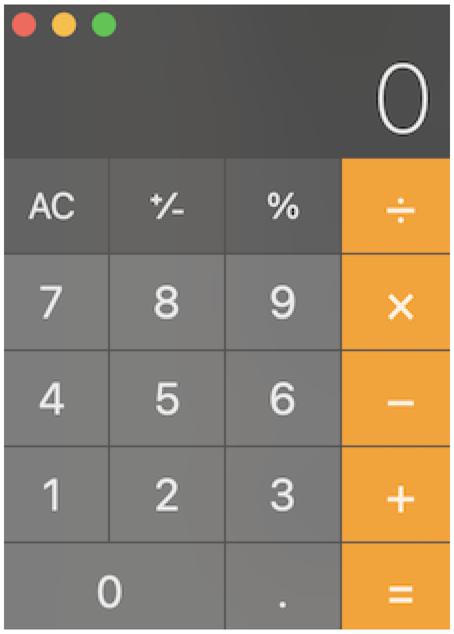
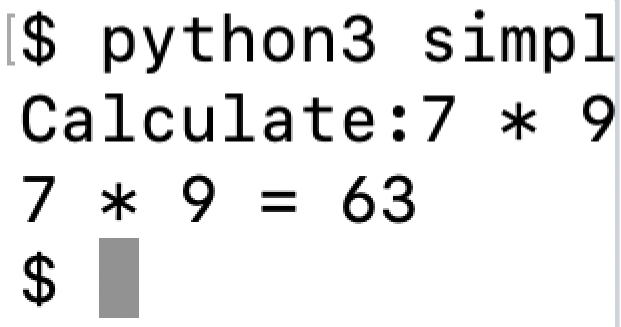
Quest

Graphics!!!!! Yep, it's time to give Python an upgrade! All the apps on your computer, everything from the calculator to the file explorer to the very browser you might be reading this in right now, are all programs that use graphics to make interactive, visual programs. You're about to get your first taste of graphics by learning how to draw shapes on the screen. Ready? By the end of this chapter you should be able to make programs that can draw!



Text vs Graphics

Up until now, the programs you have been writing are called text-based programs because the user interacts with them using the text displayed in the console. Programs that use visual features like images, colors, or buttons are graphical programs. When you set out to write programs of your own, you can ask yourself what kind of program you want to write. Some programs are better left as text-based programs, while others would benefit from a graphical component. Maybe times you get to choose, as you could use either text, graphics, or a combination of both to get the result you want.

Apple's Graphical Calculator	Text-Based Four-Function Calculator
	

```
$ python3 simpl
Calculate:7 * 9
7 * 9 = 63
$
```

Python Graphics

One of the most common libraries to create graphics in Python is called Tk (short for "tkinter"). Tk is a powerful graphics library that should be automatically installed for Windows and Mac along with Python when you installed it. While Tk's great, some of the functions are hard to use. For this reason, we provide our own small graphics library that is built on top of Tk and makes it easier to use. It's not a replacement for Tk - it just adds a few new functions to make certain things like drawing text easier. You can always explore the full Tk library if you're interested in seeing what else you can do!

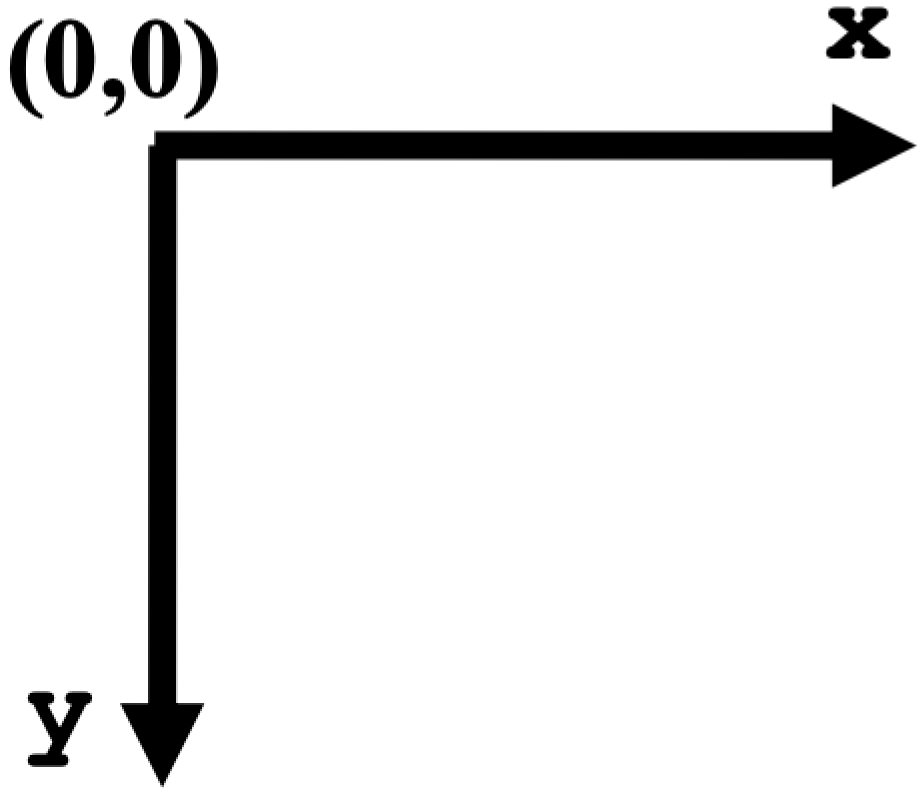
Importing

To use our graphics library, you must first import it at the top of your program, by adding this line of code to the top of your file:

```
from graphics import Canvas
```

The Canvas

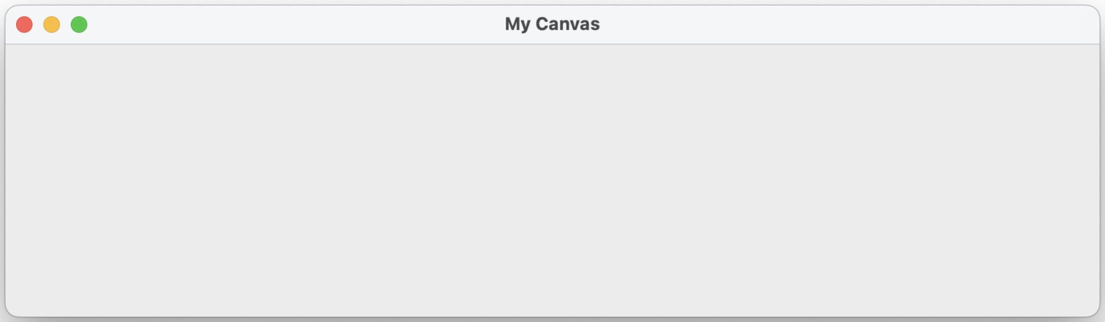
The drawing model for the graphics library is a canvas (much like a painting) where you are going to draw various shapes. The canvas is a grid of pixels that have x and y values. The coordinate (0, 0) is in the upper-left-hand corner of the canvas. The values of x increase as you move to the right. The values of y increase as you move down. In other words, you can think of the canvas as follows:



When we want to draw shapes on a canvas, we call functions on that canvas that “create” the shape we want to draw. These shapes then appear on the canvas. We will generally provide the code that creates the canvas for you, so all you need to worry about is adding shapes to that canvas. For the remainder of this handout, we will assume that a variable named `canvas` has already been created, and it represents the canvas that you'll be drawing on. Usually this will be done with the following provided code:

```
# create a new canvas
canvas = Canvas(800, 200)
```

When we create the canvas, we specify its width and height. For example, the above code will create canvas that's 800 pixels wide and 200 pixels tall



Basic Shapes

Below we provide a brief tour of some of the different shapes you can draw on a canvas as well as highlight a few of the options you have with regard to how those shapes look.

Creating Shapes:

Each time you want to draw a new shape on the screen, you use one of the various functions provided by our graphics below. They all follow a general naming convention canvas.create_<shape> where <shape> is just the name of whichever shape that particular function draws.

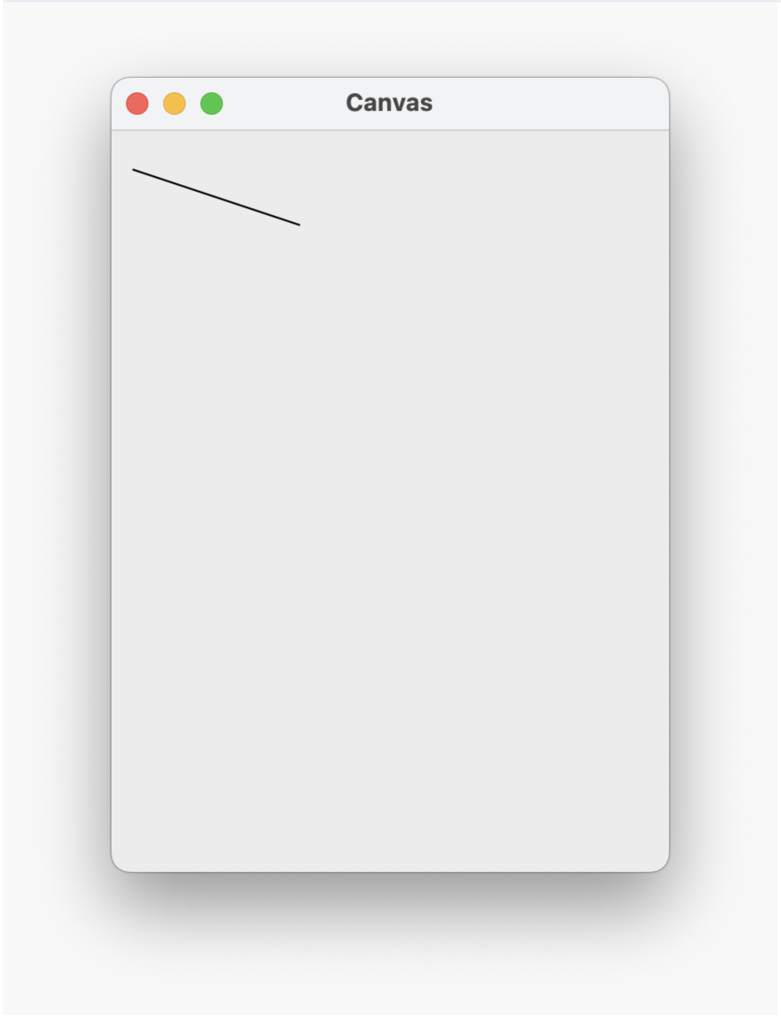
These functions do two things. They each draw some shape on the screen, but then they also return an integer associated with the order in which the shape was added to the canvas. Drawing the first shape will return an id of 1, drawing the second shape will return 2, and so on. Keeping track of these ids is a smart move in case you want to refer to that shape later on in your code (perhaps to move or delete it). While, technically, you don't have to store the shape's id in order to draw a shape, it is generally good practice to store each shape you create as a variable like so:

```
line = canvas.create_line(...) # some parameters go in here
```

Drawing Lines

To draw lines on the canvas, use `create_line` and specify the coordinates for the start and end points of the line and the line's color as a string. For example, the following command would draw a black line from location (10, 20) to (100, 50) on the canvas:

```
canvas.create_line(10, 20, 100, 50, 'black')
```

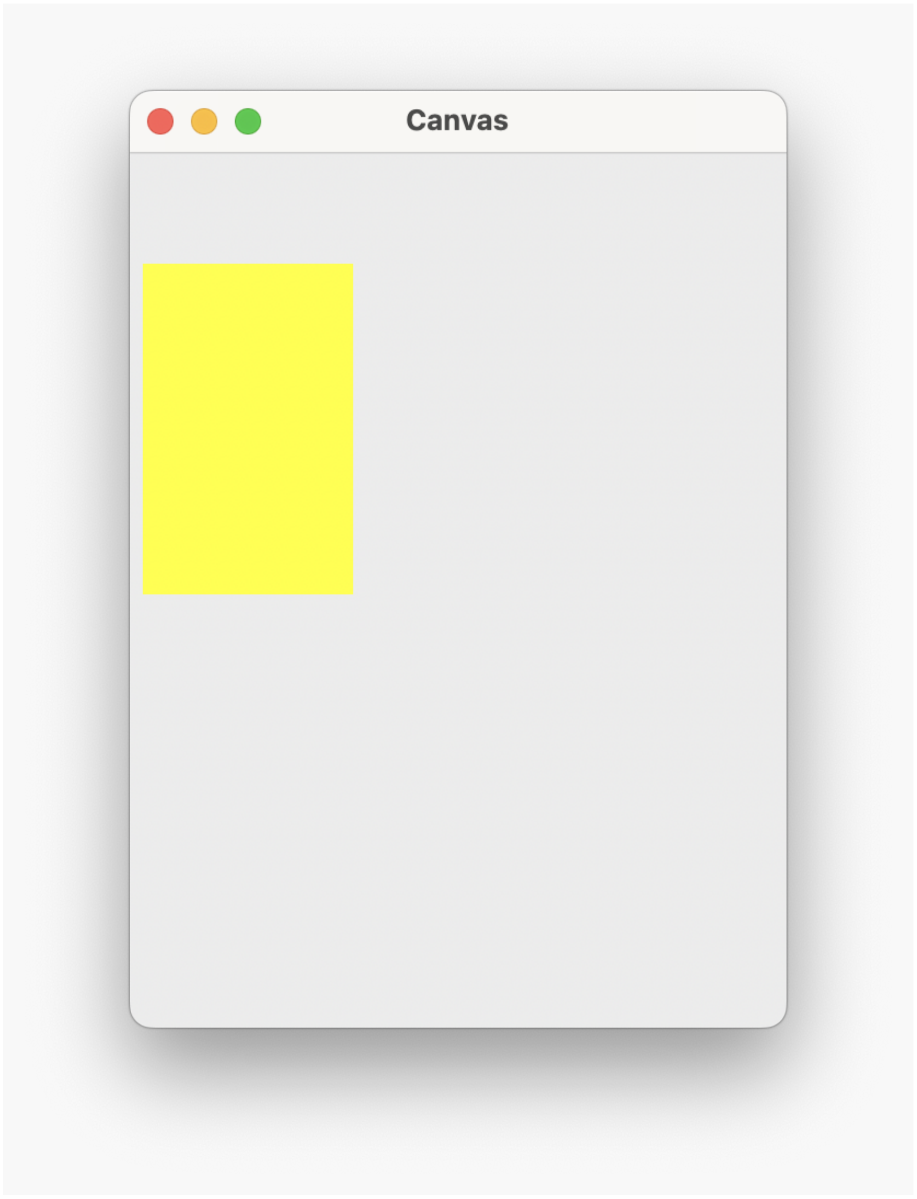


By default, all lines are drawn in black if you omit the color parameter.

Drawing Rectangles

To draw a rectangle on the canvas, use `create_rectangle` and specify the coordinates for the top left corner and bottom right corner of the rectangle, as well as a color as a string. For example, the following command would draw a yellow rectangle with upper left-hand corner at (5, 50) and lower right-hand corner at (100, 200) on the canvas:

```
canvas.create_rectangle(5, 50, 100, 200, 'yellow')
```

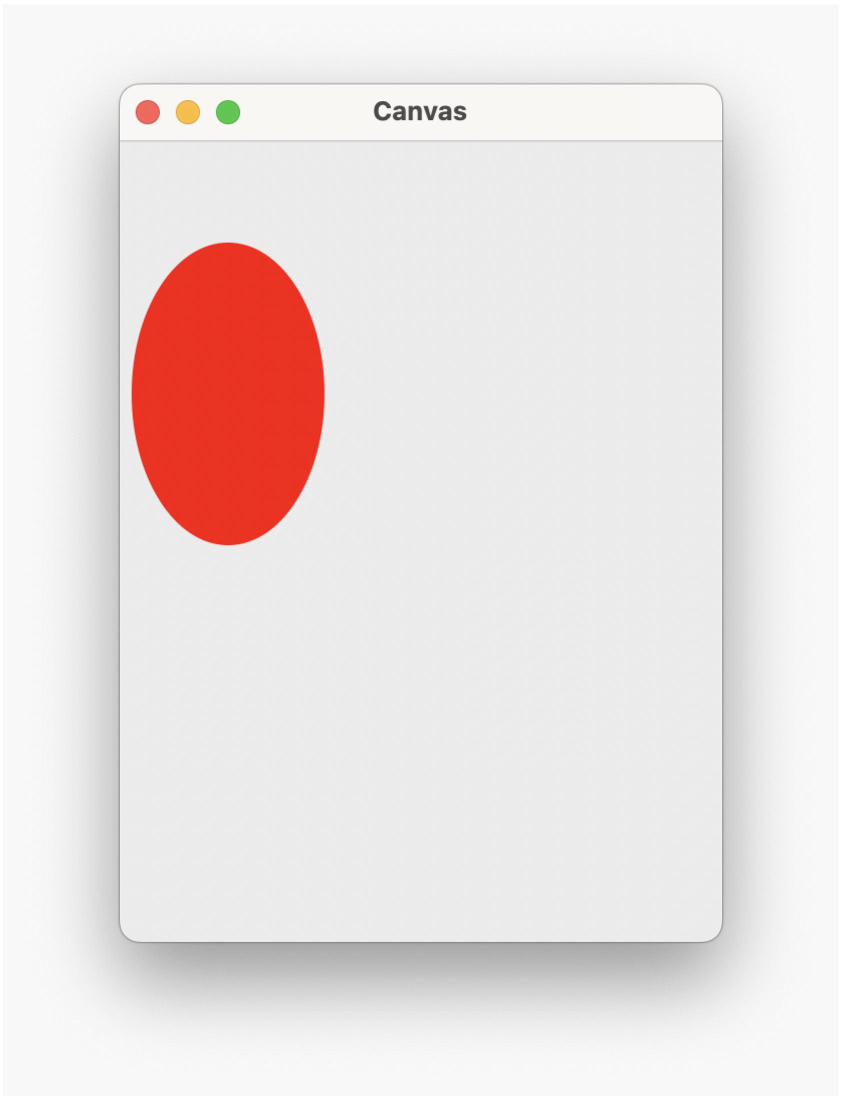


By default, all rectangles are drawn in black if you omit the color parameter.

Drawing Ovals

To draw an oval on the canvas, use `create_oval` and specify the coordinates for the top left corner and bottom right corner of the bounding box containing the oval you would like to draw. In other words, the oval drawn will have its top, bottom, right, and left just touching the sides of the bounding box you specify. You'll also specify a color as in the previous functions. For example, the following command would draw a red oval that fits within an imaginary rectangle with upper left-hand corner at (5, 50) and lower right-hand corner at (100, 200) on the canvas:

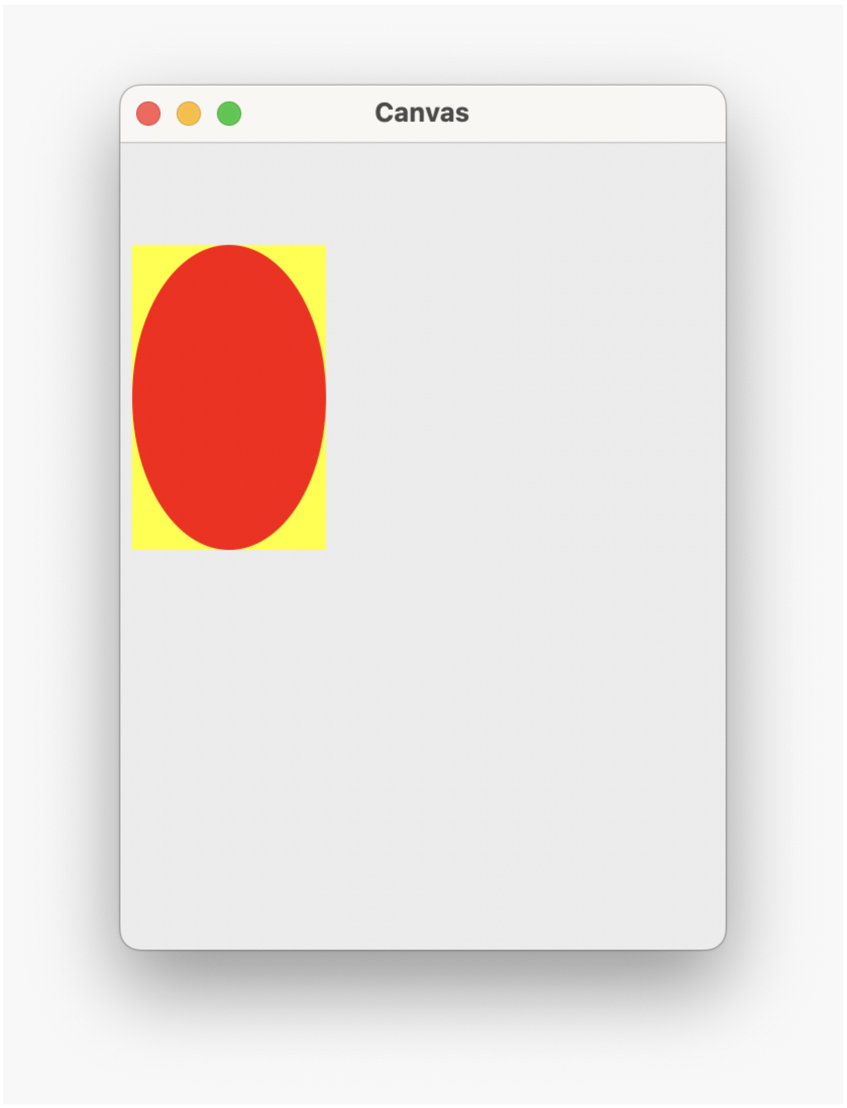
```
canvas.create_oval(5, 50, 100, 200, 'red')
```



By default, all ovals are drawn in black if you omit the color parameter.

To make clear the notion of a bounding box, below we draw the same red oval and yellow rectangle from before, where both the oval and rectangle have the same coordinates. Notice how the rectangle is like a bounding box around the oval with the same coordinates.

```
canvas.create_rectangle(5, 50, 100, 200, 'yellow')
canvas.create_oval(5, 50, 100, 200, 'red')
```



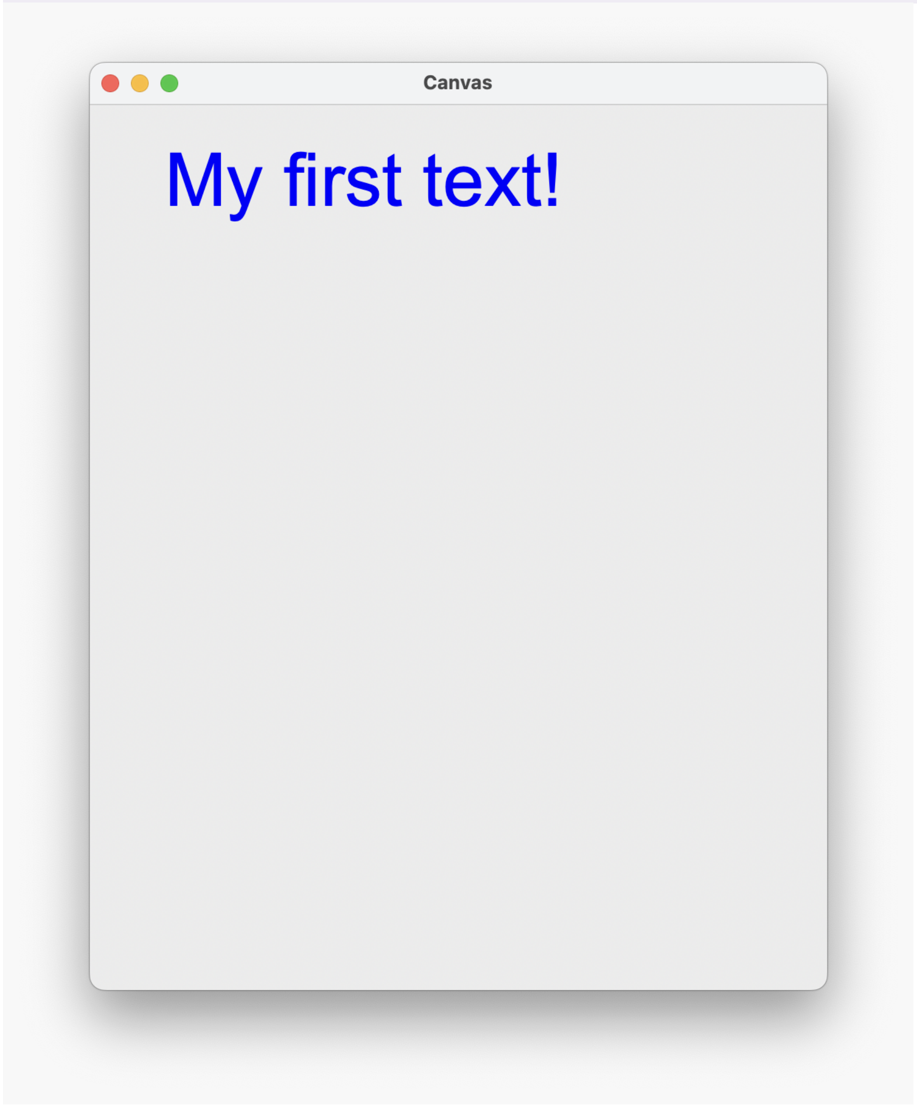
Drawing Text

To draw text on the canvas, use `create_text` to specify an x and y coordinate, the text and the following optional parameters:

- font
- font_size
- color
- anchor

The anchor is a string, one of `"n", "ne", "e", "se", "s", "sw", "w", "nw", or "center"`, that describes what side of your text you want to position at the (x, y) you provide. font describes the style and size of the text, for example 'Helvetica 32'. The text is the text string you want to appear on the screen. Here's a sample call of create_text with all of the necessary parameters (this is all one line, we've just split it to fit on the page):

```
canvas.create_text(50, 50, font='Arial', font_size = 50, text='My first text!', color='blue')
```



Removing Objects From the Canvas

You can remove objects from the canvas using delete:

```
rect = canvas.create_rectangle(100, 200, 300, 400, 'yellow')
...
canvas.delete(rect)
```

Note that deleting an object is permanent - if you want to temporarily make an object hidden, use set_hidden. You can pass in either True or False to set an object to be hidden or visible:

```
rect = canvas.create_rectangle(100, 200, 300, 400, 'yellow')
```

```
canvas.set_hidden(rect, True)  # no longer visible
...
canvas.set_hidden(rect, False) # visible again
```

Object Ordering

Objects are drawn on the canvas in the order in which they are created by your code. So, if you draw a red rectangle after a yellow circle, the red rectangle can potentially cover (occlude) part of the yellow circle. This is sometimes referred to as the z-order of the objects.

As an example, below, we first draw a yellow rectangle, then draw an orange oval (which partly overlaps the rectangle), and then some text (that is on top of both the rectangle and the oval).

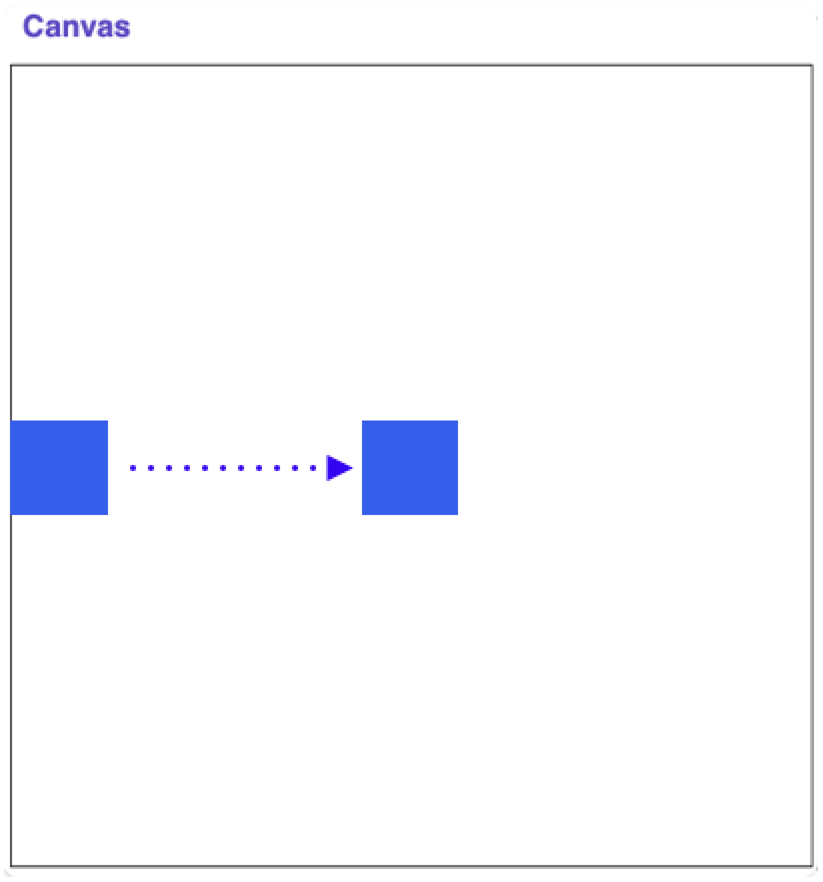
```
canvas.create_rectangle(10, 50, 200, 300, 'yellow')
canvas.create_oval(100, 100, 300, 350, 'red')
canvas.create_text(70, 170, anchor='sw', font='Arial 30', text='yay!')
```

Animation

Quest

Ok, so we know how to draw some basic shapes onto a canvas. But what if we wanted to make those shapes move? Animating shapes allows us to make really cool and dynamic graphics like the ones you see in video games. Before we reach video game level animations, let's start with the basics.

GOAL 1: Make a square move to the center.



From the last section, we know how to draw a rectangle onto the canvas:

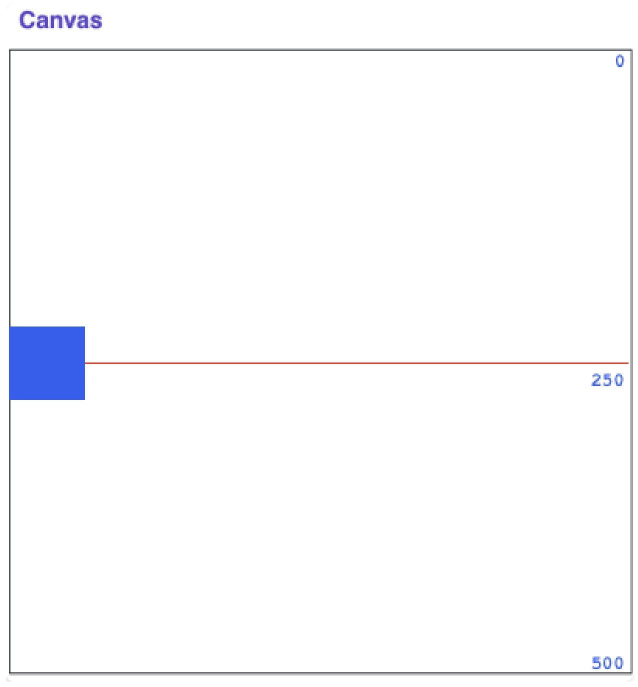
```
from graphics import Canvas

SQUARE_SIZE = 50
CANVAS_WIDTH = 500
CANVAS_HEIGHT = 500

def main():
    # makes a canvas
    canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)

    # set y coordinates
    start_y = CANVAS_HEIGHT / 2 - SQUARE_SIZE / 2
    end_y = start_y + SQUARE_SIZE

    # draw the square
    canvas.create_rectangle(0, start_y, SQUARE_SIZE, end_y, 'blue')

if __name__ == "__main__":
    main()
```

Right now, this code draws a square that is centered on the left edge. Take a minute to understand the math behind start_y and end_y.

```
start_y = CANVAS_HEIGHT / 2 - SQUARE_SIZE / 2
end_y = start_y + SQUARE_SIZE
```

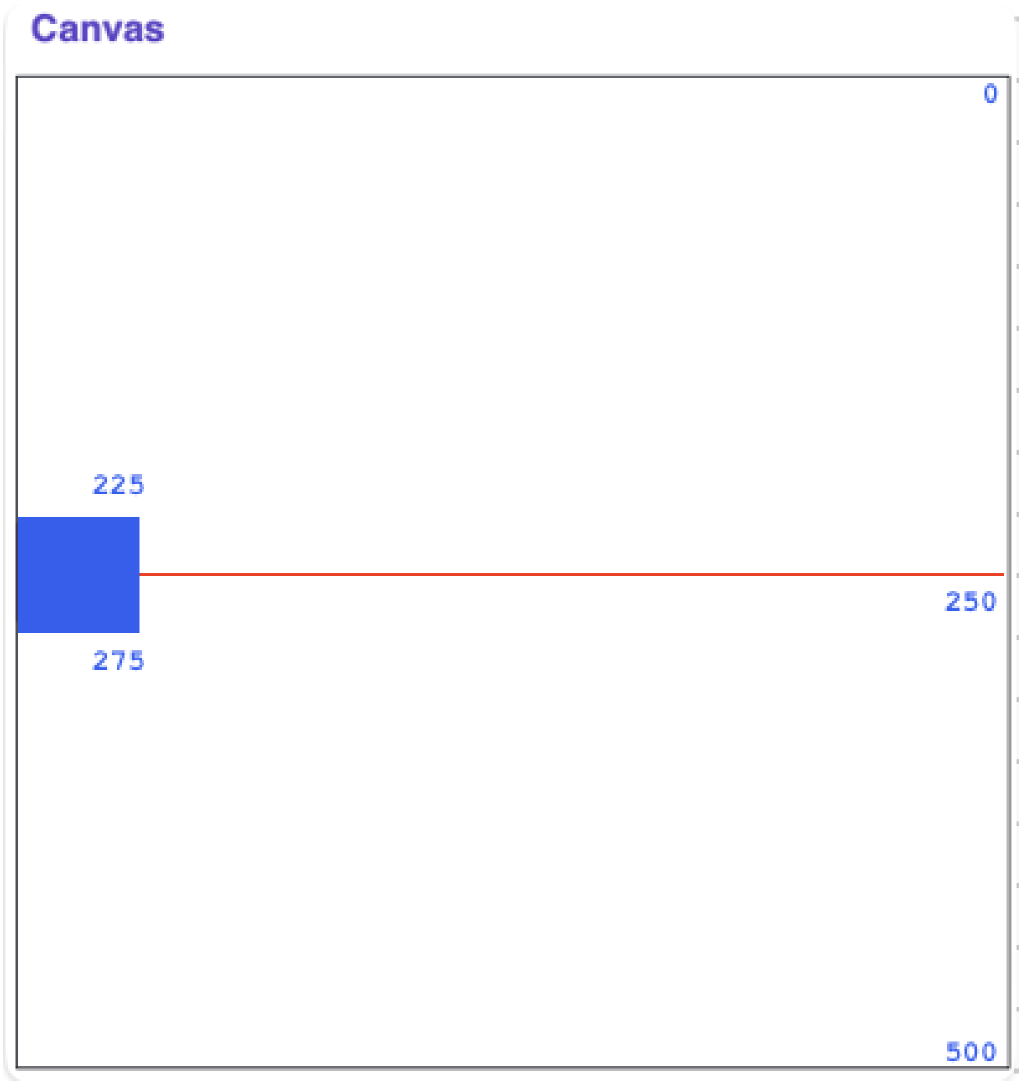
If you look at the image above, you'll see that we want the image to start center of the canvas in the y direction and on the left edge. The line that represents the center of the canvas in the y direction is the line where y = CANVAS_HEIGHT / 2.

For example, in our case, the line is where y = 250.



As you can see, the red line which represents the middle of the canvas is halfway through the blue square. If we just set start_y to CANVAS_HEIGHT / 2 then the top of the square would be the red line. To center the square on the line, we have to subtract half the square's height (essentially "moving up" the square).

end_y refers to the bottom right y coordinate. The difference between the top y coordinate and the bottom y coordinate is the height of the square, so to calculate end_y, we merely need to add SQUARE_SIZE to start_y.



Ok, now that we have the starting screen set, it's time to make it move! We want the square to move in a continuous line to the center of the screen. Or, rather, we need to make it *seem* as if the square is moving continuously. In reality, we are going to move the square to the center one pixel at a time. It will *look* continuous because we are going to increment pixels at a rate that's too fast for the human eye to pick up on.

We will call each increment or frame an update to the world. Each update will run in a loop called an animation loop. The general syntax for animation loops is found below:

```
def main()
  # setup - define all of the variables you will need
```

```
while True:
    # update world

    # pause
    time.sleep(DELAY)
```

Side note: in the section on loops, we discussed infinite loops as a bug to avoid. They can be useful for animations when we want the graphic to keep running.

Let's take a look at the last line in the program:

```
time.sleep(DELAY)
```

Here, time is a library (which means we have to import it sometime before the main() function). One function in the time library is the sleep function which pauses the program for the specified time. DELAY is a global constant also defined before the main() function which represents the number of seconds to pause the function. This pause is very necessary. Without it, the program would run so fast that we wouldn't be able to see any of the updates.

Let's implement the animation loop in our code.

```
from graphics import Canvas
import time

SQUARE_SIZE = 50
CANVAS_WIDTH = 500
CANVAS_HEIGHT = 500
DELAY = 0.001    # seconds to wait between each update

def main():
    # makes a canvas
    canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)

    # set coordinates
    start_y = CANVAS_HEIGHT / 2 - SQUARE_SIZE / 2
    end_y = start_y + SQUARE_SIZE
    cur_x = 0

    # draw the square
    square = canvas.create_rectangle(0, start_y, SQUARE_SIZE, end_y, 'blue')

    # animation loop
    while cur_x < (CANVAS_WIDTH / 2 - SQUARE_SIZE / 2):
        canvas.move(square, 1, 0)
        cur_x += 1

        # pause
        time.sleep(DELAY)

if __name__ == "__main__":
    main()
```

Let's look at all of the changes. We already discussed importing time and defining DELAY. The next new line in the code is cur_x = 0. This variable also shows up in 2 other places, but let's focus on the first two for now:

```
cur_x = 0
...
while cur_x < (CANVAS_WIDTH / 2 - SQUARE_SIZE / 2):
```

For this program we don't want the graphics to run forever. We want it to stop as soon as the square reaches the center of the canvas. To do this, we keep track of the current x coordinate of the left edge of the square cur_x and set the while loop condition to exit the loop once cur_x is a half square away from the center.

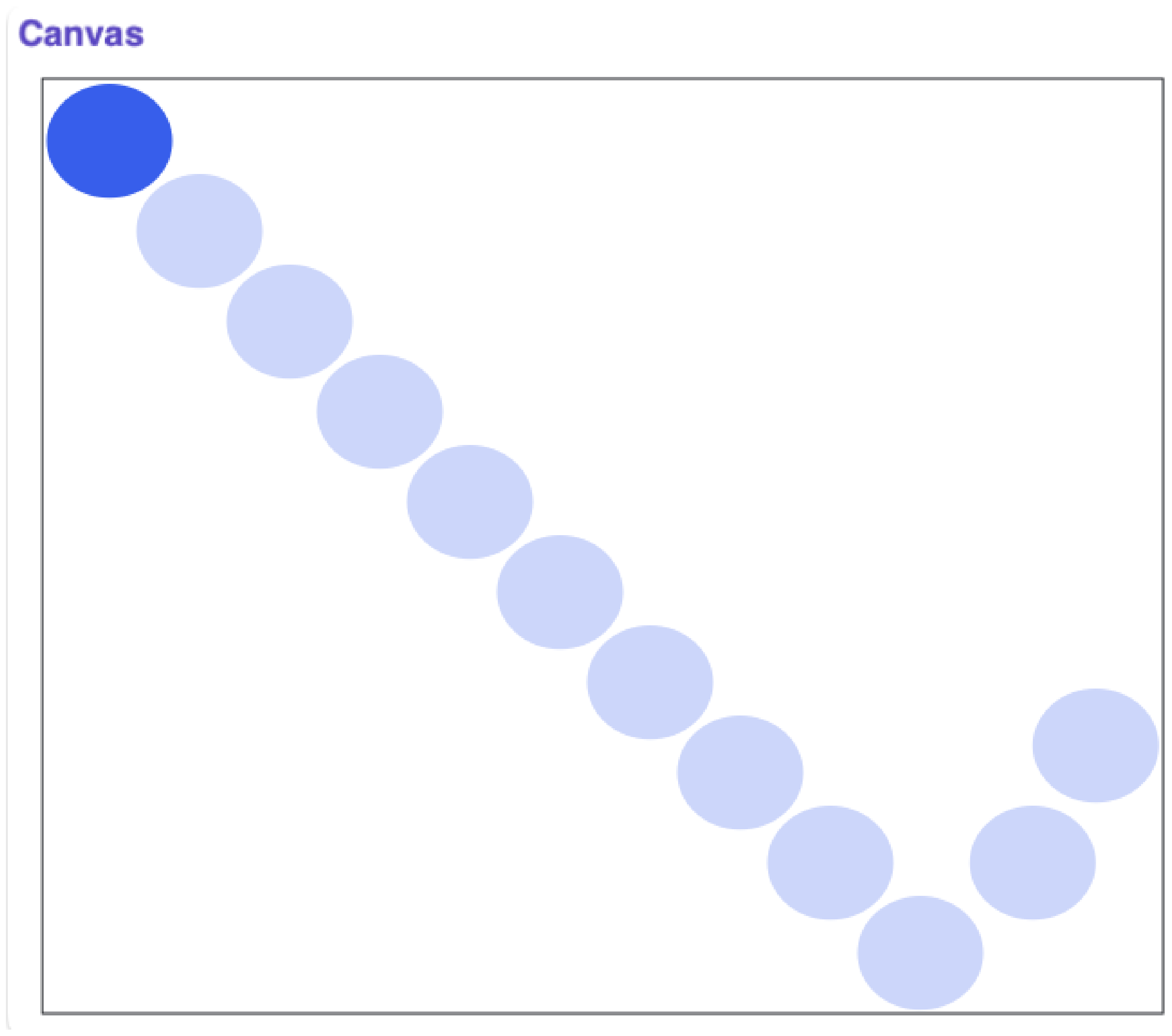
The next change big change lies in these three lines:

```
square = canvas.create_rectangle(0, start_y, SQUARE_SIZE, end_y, 'blue')

# animation loop
while ...:
    canvas.move(square, 1, 0)
    cur_x += 1
```

First, we have to assign the square that we create to a variable so that we can use it later. Then, we can use canvas's move function to move that square. The move function takes in the thing that you want to move on the canvas, the amount to move it in the x direction, and the amount to move it in the y direction. Here, we want to move to square by 1 unit to the right each iteration. Since we are moving the square, we also have to update cur_x to reflect this change.

This is the complete program. Paste it into the IDE to see it in action! Try playing around with DELAY too. The larger the value, the slower it will run. How slow can you make the program run so that it still seems continuous?

GOAL 2: Make a Bouncing Ball Program



The bouncing ball will start in the corner and move around the screen "bouncing" when it reaches an edge. Let's walk through the steps to make this happen:

Step 1: Make a "ball" display in the upper left corner of the canvas

Step 2: Make the ball move diagonally down the screen

Step 3: When the ball reaches an edge, make it change direction

Let's look at the code for Steps 1 and 2. It should look pretty familiar as it is not too different from the code in the last example:

```
from graphics import Canvas
import time
```

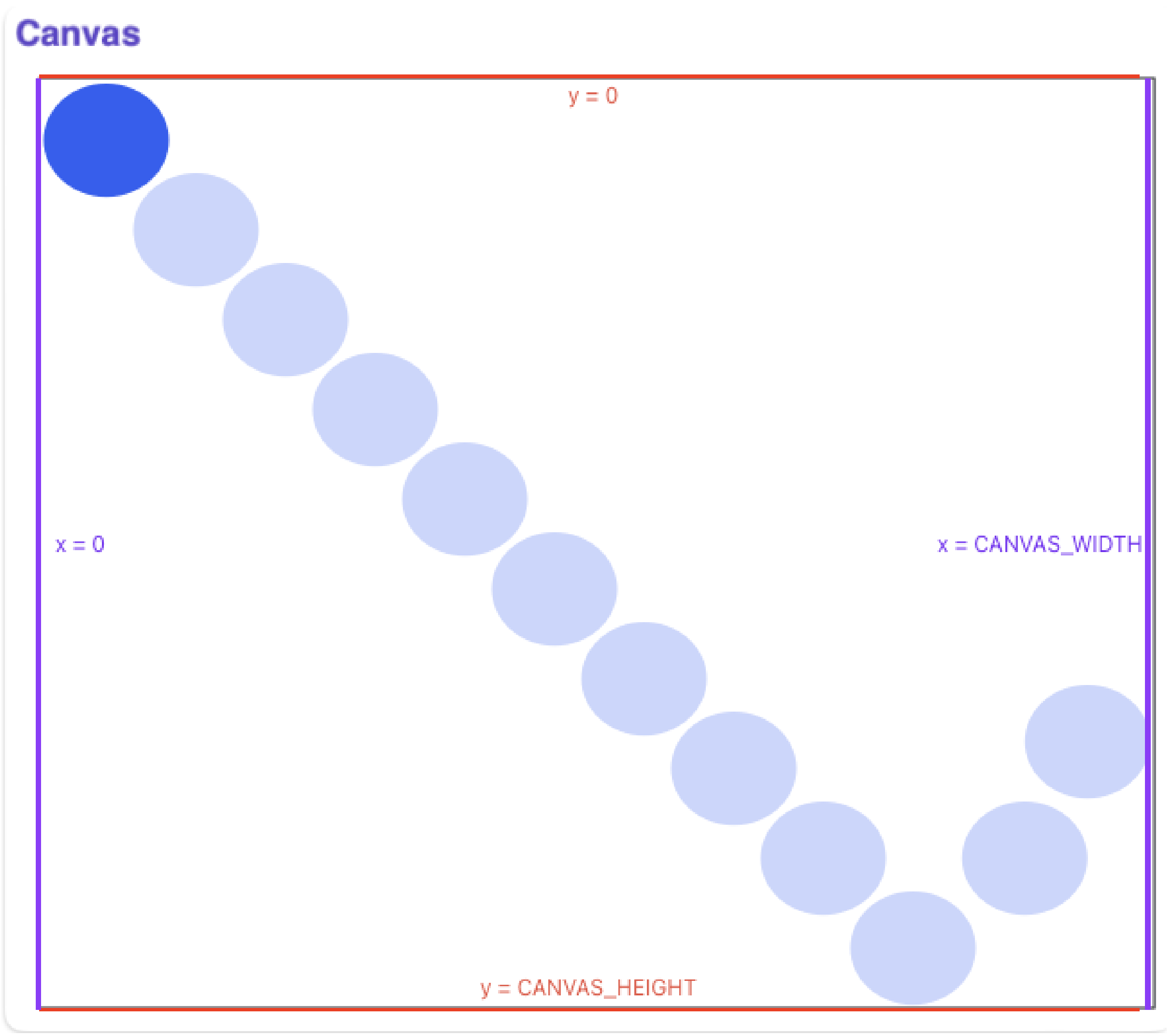
```
BALL_SIZE = 50
CANVAS_WIDTH = 550
CANVAS_HEIGHT = 450
DELAY = 0.001       # seconds to wait between each update


def main():
    # setup
    canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)
    ball = canvas.create_oval(0, 0, BALL_SIZE, BALL_SIZE, 'blue')
    change_x = 1
    change_y = 1

    # animation loop
    while(True):
        # update the ball
        canvas.move(ball, change_x, change_y)

        # pause
        time.sleep(DELAY)


if __name__ == "__main__":
    main()
```

To create the ball, we use the create_oval method. The variables change_x and change_y keep track of how much we want to increment the ball's x and y coordinates each iteration. We don't need to change those variables now, but we will in the next step.

If you paste this code into the IDE, the ball will move diagonally downwards as we want it to, but when it gets to the edge, it disappears! We need to have some way to detect when the ball reaches the edge and then change its direction.

We can track when the ball gets to the edge the same way we tracked when the square reached the middle: using variables. We'll call these variables cur_x and cur_y and they will correspond to the coordinates of the top left corner.

Next, we will need to use these variables to test if the ball has hit an edge. It's clear we need an if statement but what should the conditions be? First, it might be useful to see what the edges of the canvas are.



The top and left edges of the canvas are easy to detect. Since we are tracking the top left x and y coordinates, we can directly check if cur_x or cur_y are < 0.

```
if cur_x < 0:
    # change direction
if cur_y < 0:
    # change direction
```

What about the other two edges? Let's use the bottom edge as an example. We can't just check if cur_y is greater than CANVAS_HEIGHT. That will check if the top of the ball has passed the edge. By that point, the ball will have already disappeared. We want to check if the *bottom* of the ball has passed the edge. If you remember from the last example, the way to get the bottom y coordinate from the top is to add the height of the shape. Therefore, we need to check if cur_y + BALL_SIZE >= CANVAS_HEIGHT. The same logic can be used for cur_x and the right edge.

```
if cur_x < 0 or cur_x + BALL_SIZE >= CANVAS_WIDTH:
    # change direction

if cur_y < 0 or cur_y + BALL_SIZE >= CANVAS_HEIGHT:
    # change direction
```

Ok. Now that we have our conditions straight, how do we change directions? To go down and to the right, we set change_x and change_y equal to 1. This increases the value of the x and y coordinates by 1. This means to go up or to the left, we should decrease the values of the x and y coordinates. This would be equivalent to setting change_x or change_y to be -1. Therefore, to change or reverse directions, we should reverse the signs of change_x or change_y.

```
if cur_x < 0 or cur_x + BALL_SIZE >= CANVAS_WIDTH:
    change_x = -change_x

if cur_y < 0 or cur_y + BALL_SIZE >= CANVAS_HEIGHT:
    change_y = -change_y
```

The last thing we need to do is update cur_x and cur_y.

```
cur_x += change_x
cur_y += change_y
```

Below is the completed program. The only other thing we added were global variables for START_X and START_Y to avoid magic numbers. Once again, feel free to paste this into the IDE to see it in action!

```
from graphics import Canvas
import time

BALL_SIZE = 50
CANVAS_WIDTH = 550
CANVAS_HEIGHT = 450
DELAY = 0.001        # seconds to wait between each update
START_X = 0
START_Y = 0

def main():
    # setup
    canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)
    ball = canvas.create_oval(START_X, START_Y, BALL_SIZE, BALL_SIZE, 'blue')
    change_x = 1
    change_y = 1
    cur_x = START_X
    cur_y = START_Y
    # animation loop
    while(True):
        # change direction if ball reaches an edge
        if cur_x < 0 or cur_x + BALL_SIZE >= CANVAS_WIDTH:
            change_x = -change_x

        if cur_y < 0 or cur_y + BALL_SIZE >= CANVAS_HEIGHT:
            change_y = -change_y

        # update the ball
        canvas.move(ball, change_x, change_y)
        cur_x += change_x
        cur_y += change_y

        # pause
        time.sleep(DELAY)


if __name__ == "__main__":
    main()
```

Here is a list of a few other helpful functions that can be used to make some super cool animated programs!

```
# get the coordinates of the mouse
mouse_x = canvas.get_mouse_x()
mouse_y = canvas.get_mouse_y()

# move shape to some new coordinates
canvas.moveto(shape, new_x, new_y)

# get the coordinates of a shape
top_y = canvas.get_top_y(shape)
left_x = canvas.get_left_x(shape)

# wait for a click
canvas.wait_for_click()
```

— Functions —

Anatomy of a Function

Quest

Functions are perhaps one of the most useful programming tools we have at our disposal. They allow us to significantly shorten our programs, turning a long block of code into a single, reusable instruction. Up until now, every function you have used has been written for you. That changes today! We are now going to learn how to build functions ourselves, empowering you to write as many functions as you like and build bigger and more complex programs.

Function Call vs Function Definition

As of now, you only know how to call a function. A function call is what happens when your program runs an already defined function, like print("hello"). The thing about calling a function is you don't actually need to know how it works, you just need to know what it does. You haven't seen how print actually takes your message and writes it to the screen, but all you really need to know is that it will!

There are a lot of built-in functions in Python that you can call, but not every task you want your computer to do has been written as a function yet. That's where you come in! As the programmer, you can define additional functions that you want your program to use and then call those functions later on in the program when you need them. A function definition is exactly what it sounds like: it's what we call the part of the code that defines what a new function is going to do when we call it.

Parts of a Function

At the highest level, a function definition has three parts, a header, a body, and (optionally) a return statement. The header comes first, followed by the body, followed by the return statement:

```
def name_of_function(parameters):
    # Function Body

    # Optional Return Statement
    return value
```

Function Header

```
def name_of_function(parameters):
```

This goes at the top of every function definition, and it should actually look quite familiar. In every program you've written, you define the main function, and this code at the very bottom is actually calling the main function:

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

Python initially skips over function definitions when running a program, and it's not until the function is called that the code gets run. So, what's really going on when we hit Run on our code, is that the computer skips over all our code defined in main() and first runs the code in '__main__' where it reaches the call to main() at the bottom. Only then does it call the main function which begins our program.

If you remove the word def from the header, suddenly you've got what looks like a function call! This similarity between function calls and function headers isn't just a coincidence. The function header's whole job is to define how other parts of your program will call your function!

For starters, the def keyword tells the computer that we are defining a function instead of calling it. We then specify the name we want to use when calling the function where name_of_function is written. We give each function a unique name that differentiates it from other functions and variables. Inside of the parentheses, we have the parameters. Parameters are variables that are assigned values during the function call. We say that their values are *passed in*. This means when writing the function, we do not have to define or set the value of parameters. Parameters are useful for when we want a function to work with multiple values. We'll see many examples of this throughout this section.

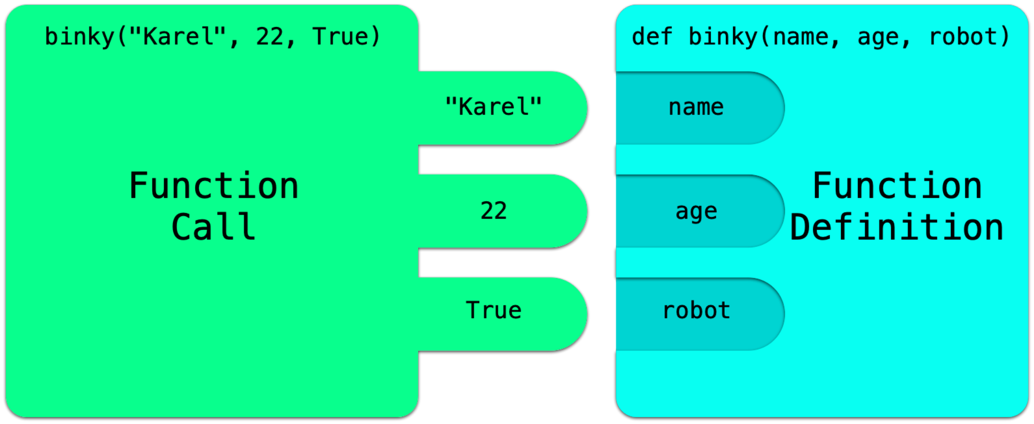
One important distinction to make is the difference between arguments and parameters. Parameters are the values in a function header/definition. Arguments are the values in a function call. Parameters are assigned the values of arguments. For both arguments and parameters, multiple values are separated by commas:

```
def name_of_function(param1, param2, param3):
```

The last step is adding a colon to the end of the header, just like you would a loop or an if statement.

There is a beautiful relationship between function calls and function definitions. They work together to allow different parts of your program to easily pass information back and forth. This makes scaling the size and complexity of your code so much cleaner and easier. For every parameter in your function definition, each function call *expects* that many arguments.

Check out this diagram comparing function headers and function calls. These two sides of programming fit together beautifully like two puzzle pieces!



Function Header Comments

A convention of good programming style is to add function header comments to the top of every function describing what the function does, any parameters it has and what it returns (if anything). Header comments look like this:

```
def name_of_function(param1, param2, param3):
    '''
    name_of_function does this
    params:
        param1 (type): what param1 should be
        param2 (type): what param2 should be
        param3 (type): what param3 should be
    return: description of what this function returns
    omit if nothing is returned
    '''
```

Headers are especially useful for other programmers trying to understand your code. It can also be helpful for ourselves when writing functions to write out exactly what each function you are trying to write is supposed to accomplish.

Function Body

```
def name_of_function(param1, param2, param3):
    '''
    name_of_function does this
    params:
        param1 (type): what param1 should be
        param2 (type): what param2 should be
        param3 (type): what param3 should be
    '''
    # Function Body
    # Function Body
    # Function Body
```

The body is the main piece of our function. It is the code we want the computer to run when the function is called. The function could carry out a computation, check that some condition is met, or perhaps just print something to the console. This part is entirely up to you. You'll use the parameters you defined in the header here, along with whatever else you want the function to do. The body is indented under the header to separate it from the rest of the program.

Function Return Statement

When a function is finished, the program returns to wherever the function was called and continues on from there. Exiting a function happens either when the program reaches the last line of the function, or when executing a return statement inside of the function. In addition to prematurely ending the function, this instruction also has the capability of sending a value back to the location of the call. You can either simply return by using the return keyword, or you can follow return with a value to additionally send that value back to the location of the call.

We're going to cover parameters and return statements in greater detail, but for now let's just look at a few examples of some functions! Below is a program that defines three different functions in addition to the main function.

```
def average(num1, num2):
   '''
   average two numbers together and return the result
   params:
       num1, num2 (int or float): numbers to average
   return: average of num1 and num2
   '''
   result = (num1 + num2) / 2
   return result
def two_goodmornings(count):
   '''
   prints "Good Morning" a max of two times
   params:
       count (int): the number of times to print "Good Morning" (max of two)
   '''
   for i in range(count):
       if i == 2:
           return
       print("Good Morning!")
# This function has no return statement
def count_to_ten():
   '''prints out numbers 1 through 10'''
   for i in range(10):
       print(i)
# This is the main function, the one we call first
def main():
   # First we'll count to ten
   count_to_ten()
   # Then we'll try to say good morning three times
   # but we can only say it twice
   two_goodmornings(3)
   # Finally, we want the average of two numbers, and
   # we're going to store the result in a variable
   num1 = 5
   num2 = 10
   my_average = average(num1, num2)

   # Now, if we print my_average, we'll see the result of
   # the 'average' function
   print(my_average)
# Below is actually the first line the program runs
# because everything else is inside of a function
# definition. Here we call the main function which
# gets our program rolling!
if __name__ == "__main__":
   main()
```

Life of a Function

You have now looked behind the scenes and learned how functions are made. We think it will be helpful to see the whole process of calling a function from start to finish. This is an exciting milestone, so you should be proud of yourself for reaching this point! See slides here.

Examples

Functions? Parameters? Return statements? We've thrown a lot at you in this section, so let's summarize and see some examples. A function has a name, it optionally takes in one or more arguments as parameters, runs some code, then optionally returns a value.

```
# 0 Parameters; no return statement
def function1():
    '''prints "Hello, World!" '''
    print("Hello, World!")
# 0 Parameters; return statement
def function2():
    '''
    prints "Hello, World!"
    return: the int 10
    '''
    message = "Hello, World!"
    print(message)
    return 10
# 1 Parameter; no return statement
def function3(message):
    '''
    prints message 3 times
    params:
        message (any): message to print
    '''
    for i in range(3):
        print(message)
# 1 Parameter; return statement
def function4(message):
    '''
    prints message
    params:
        message (any): message to print
    return: True
    '''
    print(message)
    return True
```

```
# 2 Parameters; no return statement
def function5(message, loops):
  '''
  prints message loops times
  params:
    message (any): message to print
    loops (int): number of times to print message
  '''
  for i in range(loops):
    print(message)
# 2 Parameters; return statement
def function6(name, favorite_color):
  '''
  says hi to name and states "My favorite color is Blue
  too" if favorite_color is blue.
  params:
    name (str): name of person
    favorite_color (str): favorite color of person
  return: name
  '''
  print("Hi," , name)
  if favorite_color == "Blue":
    print("My favorite color is Blue too!")
  return name

def main():
  function1()
  function2()
  function3('hi')
  function4('hello')
  function5('Coding Rocks!', 4)
  function6('Karel', 'Blue')
if __name__ == '__main__':
  main()
```

How NOT to define a function

It is very easy to make a mistake when defining a function. Just go slow, and refer back to this page if you get stuck. Below are a few common mistakes people make when defining functions:

```
# Forgot to add def at the front of the header
function1():
   print("Something is not right here...")

# Forgot to add a colon at the end
```

```
def function2()
    print("Hello, World!")

# Forgot parentheses
def function3:
    print("Hello, World!")

# Forgot commas for parameters
def function4(num1 num2):
    average = (num1 + num2) / 2

# Defined a function within a function
# Note: Technically, this is something you can do in Python
# thought it's probably not what you intended and for now
# it's best not to nest functions inside each other
def first_function():
    def second_function():
        print("Hello, World!")
```

What else are all of these functions missing? Hint: it begins with c and ends with omments!!

Scope

Quest

With your new powers of function definition, you decide to start writing a really complicated program with lots and lots of functions. You'd like to include a variable called message at the top of each function that describes what the function is doing. *Wait a second! Don't we need to use a unique name for each variable we create?* If we had to name each of these variables message_1, message_2, message_3, and so on to differentiate them, things would get messy very quickly. Even if we could come up with a clever naming system to easily identify each variable, we'd still have to use a unique word to reference each message. The more functions we add, the more complicated things get. Programmers realized this would be a problem and came up with a solution that is both efficient and adds a bit of security to our programs.

What is Scope

Scope refers to the context in which an element of your code is visible to the rest of the program. If you define a new variable, that variable name can only be used in the same function that it was defined, and that function will be called that variable's scope. Check out this program below:

🛑 Error Code

```
def count_to_n(n):
  '''
  prints numbers from 0 to n-1
  params:
    n (int): number to count up to
  '''
  print(message)
  for i in range(n):
    print(i)
def main():
  message = "Counting up from zero!"
  count_to_n(10)
if __name__ == "__main__":
  main()
```

Try running this code, and you'll see that we get an error. Congratulations, you've just encountered scope! The variable message was defined within the main function, so it is only visible within main. When we try to reference message in count_to_n, message is now **out of scope** and cannot be used.

A variable comes into scope when the program enters into the function where that variable is defined and goes out of scope when you leave that function

Why would we do this? Wouldn't it be better if you could access every variable at any time? Well, recall our example above about having many message variables in the same program. We saw that it could actually be more confusing to have all the information at once. The idea of scope is to hide all the variables that aren't relevant to what the program is currently doing. If you define 100 different functions, each with its own descriptive message variable, you probably won't need to know message_76 if you're in function_12, so Python just pretends message_76 doesn't exist.

When talking about a function, the variables that are in scope within that function are called **local variables**. Which variables are local depends on the function you are talking about, and the local variables change as the program runs through different functions. Below, you can step through a short program and watch variables come in and out of scope: See slides online in Scope section.

Reusing Variable Names

Limiting which variables we can access at different times has two remarkably useful benefits. The first is that it allows us to reuse variable names. If we want a descriptive variable at the beginning of each function we define, we don't need a hundred different variable names, because they all go out of scope as soon as we leave their respective function. Instead, we can reuse a name like message in every function, and that name will be assigned a different meaning depending on which function we are in:

```
def function1():
  message = "Running function1"
  print(message)
def function2():
  message = "Running function2"
  print(message)
def function3():
  message = "Running function3"
  print(message)
def main():
  function1()
  function2()
  function3()
```

```
if __name__ == "__main__":
    main()
```

Notice how this program prints three different strings to the console, even though we didn't change the name message across the three different functions? If scope wasn't a thing, we wouldn't be able to do this.

Function Security

A function is kind of like a restaurant. The customer comes up to the counter and orders some food, the server gives your order to the chef, the chef cooks your food, and the server gives you your meal. Similarly, a function call passes arguments to another function, that function runs some code, and then the function returns to the function call with the result.

In a restaurant, the customer doesn't go into the kitchen and play with the pots and pans, they are just there to order food. Likewise, the function that is calling another function shouldn't be poking around in the other function's body.

If your program contains a function call, the function where that call occurred is referred to as the **caller**, and the function that *gets* called is the **callee**. Consider the following program:

```
def helloworld():
    '''
    says hello to the world!
    '''
    print("Hello, world!")
def main():
    helloworld()
if __name__ == "__main__":
    main()
```

The function helloworld gets called within the main function. So, in this case, main is the caller and helloworld is the callee.

A function call is the only way the caller should interact with the callee. The arguments are the only information the callee has about the caller, and the return value is the only information the caller gets from the callee. This way, we explicitly define how information passes to and from a function, and there are no surprises.

Now we can finally talk about scope. The second reason we love scope so much is that we can leverage it to conceal variables we don't want other parts of the program to use.

Any variable defined inside of a function will be out of scope everywhere else, so it is only visible when the program is actively running that function!

Avoid Global Variables

You might be wondering what would happen if you defined a variable outside of a function. What you've created is called a **global variable**. This variable is always in scope because it was defined in the outermost part of the program.

We're actually going to pause here for a moment. Global variables are the first of a few tools you will learn about that you need to be careful about using. Although Python has no problem with you defining global variables, you should be cautious about using them and ask yourself why you have chosen to make one.

We've spent this whole section talking about the beautiful organization that scope provides us. Information is cleanly passed to and from functions and is only useful for a limited amount of time. Global variables completely throw that organization away. You should rarely, if ever, use global variables in your code, because using them is a horrible style! Global variables can cause inconsistencies within your code.

Look at the example below 🛑

```
balance = 50.0
def deposit(amount):
  '''
  deposits the amount into the bank account (balance)
  params:
    amount (number): amount of money to deposit
  '''
  balance += amount
def main():
  deposit(10.0)
  print(balance)

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

Even though balance should be in scope of the function deposit, we get an error. This is because when Python gets to the code balance += amount, it treats balance as a **local variable** to the function deposit (a variable defined within the scope of deposit) and therefore expects an assignment to have already been made in order to do the += operation.

There are ways to safely use and even modify global variables in your program, but we are just not going to teach you how. We don't want you to get into the habit of defining global variables when you really should have used local variables or passed information as arguments.

Global Constants

The only reason you should ever create a global variable is if you know with full confidence that you will never reassign a new value while the program is running. If you have a constant value that will be useful throughout your entire program, this is the time to use a global variable. We can use global constants to avoid what you might call **magic numbers**, standalone numbers that aren't assigned a name.

Let's look at an example where a global constant might actually be a good idea. In physics, there is an equation for something called gravitational potential energy. It doesn't matter what this actually means; we're just going to write a program that calculates this number using the formula

GPE = mass * gravity * height.

On Earth, **gravity** is approximately equal to **9.8 m/s2** (meter per square seconds), which we will store as the float 9.8.

Again, we are not focused on the actual physics going on here, just realize that we have this constant value 9.8 that will show up in our program.

So, you write a function called calculate_gpe that takes in the mass of an object and its height off the ground and returns the object's gravitational potential energy:

gpe.py

```
def calculate_gpe(mass, height):
    '''
    function for gravitational potential energy
    params:
        mass (number): mass of object
        height (number): height of object
    return: gpe
    '''
    return mass * 9.8 * height


def main():
    mass = float(input("What is the object's mass?"))
    height = float(input("How high off the ground is the object?"))
    print("Gravitational Potential Energy =", calculate_gpe(mass, height))
```

```
if __name__ == "__main__":
    main()
```

console

```
$ python gpe.py
What is the object's mass? 10
How high off the ground is the object? 5
Gravitational Potential Energy = 490
```

If someone else were reading this code, they might see 9.8 and have no idea why it’s there or what it represents. Assigning this value to a well-named variable would make the program much easier to read. However, a value like gravity is unique. Many formulas in physics use the value 9.8 to run calculations for objects on Earth, and we know this value is never going to be reassigned. If we wrote functions to calculate some of these other formulas, we’d have to keep defining the same variable over and over again for each function. Finally, we have a good reason to use a global variable.

When naming global constants, we capitalize them to clearly show that they are placeholders for a constant value. We would now define a global constant GRAVITY that stores the value 9.8.

gpe.py

```
GRAVITY = 9.8

def calculate_gpe(mass, height):
    '''
    function for gravitational potential energy
    params:
        mass (number): mass of object
        height (number): height of object
        return: gpe
    '''
    return mass * GRAVITY * height


def calculate_fgrav(mass):
    '''
    calculates force due to gravity
    params:
        mass (number): mass of object
    return: force of gravity on an object
    '''
    return mass * GRAVITY
```

```
def main():
    mass = float(input("What is the object's mass?"))
    height = float(input("How high off the ground is the object?"))
    print("Gravitational Potential Energy =", calculate_gpe(mass, height))
    print("Force due to gravity =", calculate_fgrav(mass))


if __name__ == "__main__":
    main()
```

console

```
$ python gpe.py
What is the object's mass? 10
How high off the ground is the object? 5
Gravitational Potential Energy = 490
Force due to gravity = 98
```

Now we actually know what that value represents in each function. The other cool thing is that every reference to GRAVITY points back to a single variable. We could reassign a new value to GRAVITY to the value of gravity on Mars and suddenly all of our functions would change too. Remember, we aren't allowed to reassign a new value to a global constant *during* a program, but it's fine to adjust the value *before* the program starts.

Parameters

Quest

Now that you have seen the basic anatomy of a function, we can dive a little deeper into each specific part. First up is parameters. In the anatomy of a function section, we defined parameters as the information that you pass into a function. Also recall that parameters are different from arguments because arguments are what goes into a function call whereas parameters go into a function definition. The purpose of this section is to further introduce you to the world of parameters so that you can properly use them to create your own functions.

Let's say you want to create a function called repeat that takes in a string s and a number n and repeats the string n times. For example, calling repeat('ab', 3) would return 'ababab'.

```
def repeat(s, n):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat('ab', 3))
if __name__ == '__main__':
  main()
```

There are several things to note from this example. First, as we can see, functions can have multiple parameters, and those parameters do not all have to have the same type. **s** is a string while **n** is an int. Similarly, the parameter types do not have to match the return type. Python does not actually check the type of parameter that you pass into a function unless you ask it to. This can be both useful and harmful depending on the type of function you are trying to write. If you want your function to work for multiple types then it's very useful, but it can lead to errors 🛑 if a user calls your function with arguments that have a type your function was not designed for:

```
def repeat(s, n):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat(4, 3))
if __name__ == '__main__':
  main()
```

If you are writing a function where the parameters have set types, and you want to indicate what types those input arguments should be, you can define that in the function header:

```
def repeat(s: str, n: int):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat('ab', '3'))
if __name__ == '__main__':
  main()
```

This will cause an error 🛑 to occur if the user tries to input a type for **s** or **n** that does not match what you have specified—like in the main function above where we pass in a string for **n** even though we have specified that it should be an int.

Note: in both of the past two examples, the program crashed. This might lead you to ask why it matters if we specify the type of the parameter in the header? Well, the answer is that oftentimes programmers don't specify. However, it can be useful for other programmers trying to read your code or if you are coming back to your code at a later time (if used in place of parameter descriptions in the function header).

Another important thing to note is that the order of arguments matters. In this case, our function expects the **s** parameter to be first. If we attempted to swap them in our function call, we would get an error 🛑:

```
def repeat(s, n):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat(3, 'abab'))
if __name__ == '__main__':
  main()
```

In addition to setting the type of a parameter, we can also add default values. A default value gives the parameter a sort of backup option to use in case no argument is given for that parameter when the function is called. If an argument is given, the argument overrides the default value and is used instead. We can see the syntax for default values below:

```
def repeat(s = 'cd', n = 4):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat('ab', 3))
  print(repeat())
  print(repeat('bc'))
if __name__ == '__main__':
  main()
```

In the last example, we see that even though only one argument was given, Python still correctly matched it to the first parameter. As we mentioned earlier, order matters. That being said, if we only wanted to change the default value for **n** there is a way to do that. We just have to tell Python which parameter we are setting in the function call:

```
def repeat(s = 'cd', n = 4):
  '''
  repeats the input string s, n times
  params:
    s (str): string to repeat
    n (int): number of times to repeat it
  return: repeated string
  '''
  repeat_str = ''
  for i in range(n):
    repeat_str += s

  return repeat_str
def main():
  print(repeat(n = 2))
if __name__ == '__main__':
  main()
```

Return vs Print

Quest

So we have these two commands print and return that keep popping up all over the place. Both of these commands take a value and spit it out somewhere, but there is a huge difference between *printing* a value and *returning* it.

Return vs Print

```
# How is this function
def meters_to_cm_case1(meters):
    return 100 * meters

# Different than this function?
def meters_to_cm_case2(meters):
    print(100 * meters)
```

The short answer is that print outputs a value to the console, while return outputs a value to a caller function.

The console is a visual tool for the programmer to see what is going on while the program is running. The console doesn't store or modify the values printed to it. The value is just pasted to the screen and that's it.

When a function returns a value, however, that value is passed back to the location of the function call as data, where it can be stored, modified, or even printed again!

If you were to run the two function above side by side, here's what would happen:

Let's run the two functions above side by side:

```
# How is this function
def meters_to_cm_case1(meters):
   return 100 * meters
# Different than this function?
def meters_to_cm_case2(meters):
   print(100 * meters)

def main():
   print('running case1')
   case1 = meters_to_cm_case1(3)
   print('running case2')
   case2 = meters_to_cm_case2(3)
   print('case1 output: ' + str(case1))
```

```
    print('case2 output: ' + str(case2))

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

Here's what's happening:

- The first function doesn't print anything to the console, but the value 100 * meters is be returned to the caller for later use (case1).
- The second function prints the value of 100 * meters to the console, but then the program just exits and the value we printed is be lost (which is why None is the value of case2).

This brings up another important difference. print does not end a function but return does! We can put multiple print statements back to back, and all of them will print something to the console. However, if we stacked several return statements on top of each other, only the first one would be executed.

```
# How is this function
def multiple_returns():
    return "Howdy!"
    return "Howdy!"
    return "Howdy!"
# Different than this function?
def multiple_prints():
    print("Hey there!")
    print("Hey there!")
    print("Hey there!")
def main():
    multiple_returns()
    multiple_prints()
if __name__ == '__main__':
    main()
```

Even though multiple_returns() returns a string, we never actually store or print that value so it is lost. On the other hand, multiple_prints() does print stuff to the console, but it doesn't return anything. This is why we don't see any "Howdy!" messages in the console.

Note: multiple returns are tricky. In the multiple_returns() function, the last two returns are actually not reached because each function can only return once. We will talk more about this in the next section.

Functions always return

We use return for two reasons: to stop a function early and to return information. If your function doesn't need to do either of these things, you won't need to use the keyword return. Still, even without a return statement, your function does still return no matter what. If the program ever reaches the last line of a function, it will automatically return to wherever that function was called. print does not work this way! A function is not guaranteed to print anything to the console. You have to specifically write a print statement to make that happen.

Maybe a cool infographic showing the "pipeline" of return vs. print with directional pipes connected to the caller or the console

Multiple Returns

Quest

Welcome to part 2 of our saga on return statements. We know what return statements are (statements that give back information from the function) and we know how to distinguish them from print statements. So what is left to learn? Well, as the title of this section suggests, it is possible to have multiple return statements in a function. WHAT?! 😯 How is this possible? Let's see below.

```
def get_max(num1, num2):
  '''
  gets the maximum of the two input numbers
  params:
    num1 (int or float): the first input number
    num2 (int or float): the second input number
  returns: the maximum of num1 and num2
  '''
  if num1 > num2:
    return num1
  return num2
def main():
  print(get_max(4, 7))
if __name__ == '__main__':
  main()
```

This code is completely valid. Let's discuss why. Based on what we know about if statements, any code not in an attached else statement should also run after the body of the if has finished executing (given that the condition was true). So why doesn't the second return statement run? As we've said before, return statements end the function.

This means that if the conditional is true and return num1 runs, everything after that will not be executed. We never get to return num2 even though it is not in an else.

Let's look at another example. This example doesn't have multiple return statements in code, but it has a single return statement within a for loop which runs multiple times. What will this do?

[⚠️ Buggy Code]

```
def count_to_num(num):
  '''
  counts from 1 to num
  params: num: the number to count to
  returns: the counted numbers
```

```
    '''
    for i in range(1, num+1):
        return i
def main():
    print(count_to_num(5))
if __name__ == '__main__':
    main()
```

Huh. That's not what we wanted. We wanted our function to count all the way from one to the inputted number but it only returned the first number. Even in for loops, the return statements end the function. The rest of the for loop is not executed, so the only number that is returned is 1. In this particular case, the issue can be fixed by changing the return statement to a print statement.

```
def count_to_num(num):
    '''
    counts from 1 to num
    params: num: the number to count to
    returns: the counted numbers
    '''
    for i in range(1, num+1):
        print(i)
def main():
    count_to_num(5)
if __name__ == '__main__':
    main()
```

This solution does not actually lead to returning multiple things but avoids having to do that all together. We will see a few ways to return multiple things in later sections. Below is another example of how ***not*** *to return multiple things:*

[⚠️ Buggy Code]

```
def divisible(num1, num2):
    '''
    checks to see if num1 is evenly divisible by num2 (assuming that num1 is
    larger than num2)
    params:
        num1 (int): dividend (number being divided)
        num2 (int): divisor (number dividing into num1)
    returns: true and the quotient if the number goes in evenly, false and the
    remainder otherwise
    '''
    divides_evenly = False
    result = num1 % num2
    if result == 0:
        divides_evenly = True
        result = num1 / num2
    return divides_evenly
    return result
def main():
    print(divisible(18, 6))
if __name__ == '__main__':
    main()
```

As you can see, no error occurs, but the second return statement is completely ignored. If statements and conditionals allow for multiple return statements, but they cannot just exist side by side.

Function Decomposition

Quest

We have come to the end of our two section saga on return statements, and now, we are ready to put it all together to build a program! Well… almost ready. There is one last concept that we want to touch on before we are ready to start testing and deploying programs on our own. We already saw how useful function decomposition can be in Karel with functions like fill_pothole(), turn_around(), and turn_right(). This concept is *so* useful and important that we thought we'd show you function decomposition in Python as well (and maybe get to make some cool art while we do it).

As a small reminder: function decomposition is the process of breaking a program into smaller subproblems that make the code easier to read and understand. These subproblems are usually broken down by creating a separate function which the main function can then call. This is especially useful in cases where we have a program that repeats code very frequently. In Python, these subproblem functions are called helper functions because they help the main program accomplish its task.

Ok, now that we've reviewed and added some definitions, let's look at an example.

Worked Example - IO Art

Now it's time to make some art! The program below is designed to make "io art" which is basically a fun way of saying it prints the letters i and o with varying spaces in between. If we vary the number of spaces in the right way, we can make "diamonds." Run this code to see it in action!

```python
def main():
   MAX_SPACES = 20
   DIAMONDS = 3

   for i in range(MAX_SPACES):
      print(" ", end="")
   print("io")
   for i in range(DIAMONDS):
      for j in range(1, MAX_SPACES):
         for k in range(MAX_SPACES-j):
            print(" ", end="")
         print("i", end="")
         for k in range(2 * j):
            print(" ", end="")
         print("o")
      for j in range(MAX_SPACES + 1):
         for k in range(j):
            print(" ", end="")
         print("i", end="")
         for k in range(2 * (MAX_SPACES - j)):
            print(" ", end="")
         print("o")
if __name__ == "__main__":
   main()
```

This program is pretty cool, but right now, reading it is a bit of a headache. That's because we tried to fit this program into one function! We clearly need to decompose it. Let's look at the decomposed version below.

```
def print_n_spaces(n):
   '''
   print_n_spaces prints a row of n spaces on the same line
   params: n (int): number of spaces to print
   '''
   for i in range(n):
      print(" ", end="")  # end="" makes sure the spaces are on the same line
def print_io_line(gap, max_size):
   '''
   print_io_line prints i and o on a line with a certain number of spaces in
        between

   params:
       gap (int): 1/2 the number of spaces in between i and o
       max_size (int): the maximum number of spaces in a row
   '''
   # Offset io from front of line
   print_n_spaces(max_size - gap)

   # Seperate i and o
   print("i", end="")
   print_n_spaces(2 * gap)
   print("o")
def print_diamond(max_size):
   '''
   print_diamond prints a diamond of io lines where the
   gap in between the i and the o increases until it
   reaches max_size and then decreases to 0
   params:
      max_size (int): the maximum number of spaces in between i and o at
      any one point
   '''
   # Increase gap to max_size
   for i in range(max_size):
      print_io_line(i, max_size)

   # Decrease gap back to 0
   for i in range(max_size + 1):
      print_io_line(max_size - i, max_size)
def main():
   # How wide should the diamonds be?
   max_size = 20

   # How many diamonds do you want?
   DIAMONDS = 3

   # Print each diamond
   for i in range(DIAMONDS):
      print_diamond(max_size)

if __name__ == "__main__":
   main()
```

Look at how much easier this is to read! The main function alone is now only 4 lines of actual code, and it is easy to understand what it does. You can look at the function header comments to see what each function does. Try to look through each helper function and see how the breakdown of functions works. Which functions are calling which other functions? What's the order of each function call? Understanding this is the key to what makes decomposition so useful.

Catching a Bug

Say for example we had a bug in our code where we forgot to add a plus one (also called an **off by one** bug ⚠️). In the undecomposed code, this would look like this:

```
def main():
    MAX_SPACES = 20
    DIAMONDS = 3

    for i in range(MAX_SPACES):
        print(" ", end="")
    print("io")
    for i in range(DIAMONDS):
        for j in range(1, MAX_SPACES):
            for k in range(MAX_SPACES - j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * j):
                print(" ", end="")
            print("o")

        for j in range(MAX_SPACES): # deleted the + 1
            for k in range(j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * (MAX_SPACES - j)):
                print(" ", end="")
            print("o")
if __name__ == "__main__":
    main()
```

If you run this code, you will see that the ends of most of the diamonds are missing! From the code above, without the comment there to show where the error is, would you have been able to figure out which line of code was causing the error? You probably could have (we have faith in you), but it could have taken a *really* long time. With decomposition, this bug is much easier to figure out. Consider the same bug in the decomposed code below:

```
def print_n_spaces(n):
    '''
    print_n_spaces prints a row of n spaces on the same line
    params: n (int): number of spaces to print
    '''
    for i in range(n):
        print(" ", end="")
def print_io_line(gap, max_size):
    '''
    print_io_line prints i and o on a line with a certain
    number of spaces in between

    params:
        gap (int): 1/2 the number of spaces in between i and o
        max_size (int): the maximum number of spaces in a row
    '''
    # Offset io from front of line
    print_n_spaces(max_size - gap)
    # Separate i and o
    print("i", end="")
    print_n_spaces(2 * gap)
    print("o")
def print_diamond(max_size):
    '''
    print_diamond prints a diamond of io lines where the gap in between the i
    and the o increases until it reaches max_size and then decreases to 0
    params:
        max_size (int): the maximum number of spaces in between i and o at any
        one point
    '''
    # Increase gap to max_size
    for i in range(max_size):
        print_io_line(i, max_size)

    # Decrease gap back to 0
    for i in range(max_size):    # <- off by one error here
        print_io_line(max_size - i, max_size)
def main():
    # How wide should the diamonds be?
    max_size = 20

    # How many diamonds do you want?
    DIAMONDS = 3

    # Print each diamond
    for i in range(DIAMONDS):
        print_diamond(max_size)
if __name__ == "__main__":
    main()
```

Here, the bug is in the print_diamond method. If we are using good programming and debugging practices, we are testing all of our functions and helper functions separately in addition to testing the program as a whole. This way, we could use our separate tests to determine that the helper functions print_n_spaces and print_io_line work as intended. In fact, when we tested print_diamond and tried to print a single diamond, we would find that the error occurs here (and not in main). print_diamond is four lines of actual code. Through decomposition (and good testing practices), you can narrow down the number of lines of code you have to look through in this program to find a bug from 26 to 4. As you write larger and more complex programs, this will become even more necessary.

Benefits of Function Decomposition

As you can see from the IO example, while function decomposition is not a *requirement* to have working code, it makes code much easier to read and understand which is an important part of programming as well. Less repetition of code also makes it easier to debug and harder to make mistakes. If there is a section of your code that appears to be buggy, you can test a particular helper function rather than having to look at the program as a whole. This can save *hours* when debugging. Trust us. Good style is always good practice.

Once you understand how this program is set up, you can play around with it and try to make your own io art!

Update Functions

Quest

Sometimes, after storing a value in a variable, you want that value to change as your program runs. One example of this is the variable `i` in our for-loops! At a given point, `i` stores the value for which iteration of the loop is being run. After each iteration, `i` is increased by `1` to keep track of how many times the program has looped. This concept of setting and then updating a value is a common one, and we are going to learn about a neat conceptual way of writing code to make that process happen.

Let's play a game! We are going to write down a sequence of numbers starting with a positive integer that you pick. Go ahead, pick one now! The next number in your sequence depends on a few simple rules:

1. If the current number is `1`, then the sequence ends
2. If the current number is *even*, then the next number is `n / 2`
3. If the current number is *odd*, then the next number is `(3 * n) + 1`

We apply these rules to each number in the sequence until the sequence finally ends on a `1`. As an example, say we picked the number 5 to start:

- `5` is odd, so the next number is `(3 * 5) + 1 = 16`
- `16` is even, so the next number is `16 / 2 = 8`
- `8` is even, so the next number is `8 / 2 = 4`
- `4` is even, so the next number is `4 / 2 = 2`
- `2` is even, so the next number is `2 / 2 = 1`
- `1` is the last number of the sequence

So, the final sequence is `[5, 16, 8, 4, 2, 1]`.

Go ahead and try this out with the number you picked!

Could we write a program to do this for us? We know how to ask for user input, handle different cases with conditionals, and loop over code multiple times. We should be able to use our Python skills to get the job done!

First, we start with some pseudocode. This is always a great first step when figuring out how to structure your code at a high level:

PSEUDOCODE

```
number = some a positive integer
repeat until number = 1:
       write down that number
```

```
    number = the next number in the sequence
write down a 1
```

x = update(x)

The magic really happens in the line where we updated the value of `number`:

```
number = the next number in the sequence
```

We could accomplish this step with a helper function that takes in the current number and outputs the next number according to the rules we stated above! Let's call this function `next_number`:

```
number = next_number(number)
```

This instruction sets the value of `number` equal to the result of feeding `number` into `next_number`. We refer to this type of instruction as an `x = update(x)` instruction because `number` is both the argument passed into `next_number` and the variable where the result of `next_number` is stored. We decided to name functions like `next_number` update functions because we use them to update the value of a variable according to some rules.

How do we want `next_number` to work? Recall your skills in writing new functions and think about what this function needs to do:

Done thinking? Ok, let's look below:

```
def next_number(number)
    '''
    next_number takes in the current number of a sequence and determines what
the next
    number in the sequence should be
    params:
        number (int): previous number in the current sequence
    return: (int): next number in the sequence
    '''
    # If number is 1 then return a zero to indicate the sequence has ended
    if number == 1:
        return 0
    elif number % 2 == 0:  # If number is even, the next number is number/2
        return number / 2
    else: # If number is odd, the next number is (3 * number) + 1
        return (3 * number) + 1
```

In this implementation, we chose to use a 0 to indicate to the program that the sequence should end, but how you go about doing that is entirely up to you!

Update functions are useful when the change you want to apply to a value is more complicated than what can be accomplished in a single line. In the case of `next_number`, the result depends on whether the number passed in is 1, even, or odd and requires multiple if-statements to handle each of these cases. This way, we decompose all that extra code into a singular, easy-to-read function.

Update functions aren't explicitly a feature of Python, but they are a concept built from tools Python does give us, like variables, functions, and parameters. And they're super useful too!

Fun fact, these sequences come from a real mathematical concept called Hailstone Sequences. Mathematicians are still trying to figure out whether hailstone sequences always terminate at 1, or if there are some integers that result in an infinite hailstone sequence that never ends.

Curious about these peculiar sequences? Read more about this fascinating unsolved problem: https://en.wikipedia.org/wiki/Collatz_conjecture

— Guides —

Python Documentation

Getting Started

Every program you write for Code in Place will always include the following code to define and execute a `main()` function:

```
def main():
    # Your code goes here

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

Variables

Assigning a Variable

```
variable_name = value
```

Accessing the Current Value of a Variable

```
message = "Hello, World!"

# message is replaced with its current assigned value, which is "Hello, World!"
print(message)
```

Variable Types

Type	Stores	Example
`int` (integer)	Whole numbers	137
`float` (floating point)	Decimal numbers	137.0
`str` (string)	text/character sequence between `""` or `''`	“Hello, World!” *Note: The quotes won't appear when printing a string*
`bool` (boolean)	True or False	True

Print

Printing a string literal

```
# Remember to surround your message in single '' or double "" quotes
print("Hello, World!")
```

Printing a variable

```
# Change the value of data and see what happens
data = 137.2
print(data)
```

Multi-Argument Print:

```
name = "Karel"
# Python automatically separates each argument with a space
print("Hello, my name is", name)
```

Input

Storing user input as a variable

```
user_input = input("This Message Will Is Displayed When Prompting The User For Input")
```

Storing numerical inputs

```
# input always returns a string by default, so you have to convert numerical inputs
manually
integer_input = int(input("Enter an integer:"))

float_input = float(input("Enter a floating point value:"))
```

Operators

Arithmetic Operations

Operation	Symbol	Example

Addition	+	`>>> 20 + 11`
		`31`

Subtraction	-	`>>> 17 - 23`
		`-6`

Multiplication	*	`>>> 15 * 12`
		`180`

Division	/	`>>> 10 / 2`
		`5.0`

Integer Division	`//`	`>>> 10 // 4`
		`2`

Exponentiation	`**`	`>>> 2 ** 3`
		`8`
		`>>> 2 ** -1`
		`0.5`

Modulus	`%`	`>>> 8 % 3`
		`2`

Negation	`-`	`>>> x = 0.5`
		`>>> -x`
		`-0.5`

Comparison Operations

Comparison Operator	Meaning	True Example	False Example
`==`	Equal	`1+1 == 2`	`2 == 3`
`!=`	Does not equal	`3.2 != 3`	`5-5 != 0`
`<`	Less than	`5 < 7`	`2 < 1`
`>`	Greater than	`4 > 2`	`1 > 9`
`<=`	Less than or equal to	`100 <= 90`	`4 <= 18.4`
`>=`	Greater than or equal to	`5.0 >= 5.0`	`11 >= 20`

Logical Operations

Operator	True Example	False Example
`and`	`(3 < 5) and (-1 == -1)`	`(7 == 8) and (12 > 2)`
`or`	`(0 == 1) or (10 <= 15)`	`(0 > 5) or (6 == 3)`
`not`	`not False`	`not 1 > 0`

Writing expressions with variables

Expressions are evaluated before assignment. The right side of the = becomes a single value which then gets assigned to the variable name.

Example Math Expression

```
age = 22
age_after_birthday = age + 1
print(age_after_birthday)
```

Example Logical Expression

```
silly = True
funny = True
mean = False
if funny and silly:
   print("Nice to meet you!")
if not mean:
   print("Thank you!")
```

Order of Precedence

Highest

() parentheses
** exponentiation
– negation
* / // %
+ –
< > <= >= == !=
not
and or

Lowest

Conditionals & Loops

If Statements

```
if condition:
   # Do something
```

While Loop

```
while condition:
    # Repeat this until condition is false
```

For Loop

```
for i in range(n):
    # Repeat this n times
```

Debugging Tips

Here we are going to list the some common errors you might see while writing code for this class as well as give you a few tools in your tool belt for debugging buggy code.

Common Errors 🛑

🛑 `SyntaxError: EOL while scanning string literal`

```
def main():
    print("Hello, world!')
if __name__ == '__main__':
    main()
```

This error occurs when you forget to close the quotations of a string (or accidentally mismatch a single quote with a double).

🛑 `SyntaxError: cannot assign to literal`

```
def main():
    22 = age
if __name__ == '__main__':
    main()
```

This error occurs when you accidentally put the variable name and variable value on the wrong side. Remember: the variable **name** goes on the left side and the variable **value** (some expression) goes on the right.

🛑 `NameError: name '[variable name]' is not defined`

```
def main():
    age
if __name__ == '__main__':
    main()
```

This error occurs when you try to use a variable before you have given it a value or if you try to use a variable outside of its scope. This can also occur if you forget the quotations for a string.

🛑 `ZeroDivisionError: division by zero`

```
def main():
    x = 51 % 17
    y = 10
    print(y / x)
```

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

Most times you see a zero division error, it's not because you explicitly wrote `number / 0`. Most times it's because you have set some variable equal to an expression that you didn't realize evaluates to `0`. When you run into this error, be sure to thoroughly inspect the right side of the assignment operator to see why it evaluates to `0 (`for example, it could be due to a floating point error if your code involves small numbers`)`. Note: this error is essentially the same as:

`ZeroDivisionError: integer division or modulo by zero`

🛑 `TypeError: can only concatenate str (not "int/float") to str`

```
def main():
   x = '80' + 16.0
   print(x)
if __name__ == '__main__':
   main()
```

This error occurs when you try to concatenate a string and a number. Before you can concatenate two the strings or add the two numbers, you have to use explicit type conversion to convert one of them to the desired type.

🛑 `TypeError: unsupported operand type(s) for [operator]: '[type]' and '[type]'`

```
def main():
   x = input('enter a value: ')
   print(x / 2)
if __name__ == "__main__":
   main()
```

This occurs when you try to do some operation between two types that can't support the given operation (example: you can't divide a string by an int). This is similar to the error above with concatenation. Especially if the input function is involved, do not forget to convert types.

🛑 `ValueError: math domain error`

```
import math
def main():
   i = math.sqrt(-1)
   undefined = math.log(0)
```

```
    print("this can't be right")
if __name__ == '__main__':
    main()
```

When using the math class, you must pay attention to math domain rules. Inputs to `math.sqrt()` must be non-negative. Inputs to `math.log()` must be positive.

Common Bugs ⚠️

These are some common bugs that you might see in your code that aren't necessarily errors (i.e. won't cause an error to print to the console) but will still make your code not work the way that you want it to.

Bug ⚠️: incorrectly swapped variables

```
def main():
    a = "Chris Piech"
    b = "Mehran Sahami"
    a = b        # a now points to "Mehran Sahami"
    b = a        # b still points to "Mehran Sahami"
    print(a)
    print(b)
if __name__ == '__main__':
    main()
```

Fix: use a temp variable

Bug ⚠️: infinite loops

```
def main():
    i = 0
    while i < 1:
        i = i - 1
if __name__ == '__main__':
    main()
```

This code doesn't necessarily error but it will keep running forever! We don't want that to happen. Make sure whatever condition you put in the loop is a condition that can end at some point. Most programs that you will write in this class should not take too long to run so if a program takes more than a few seconds, try checking for an infinite loop.

Bug ⚠️: putting code in the wrong loop (inner vs outer)

```
def main():
    number = 64
    while (number >= 1):
      for i in range(3):
         print("I'm going to count to three!")
         print(i+1)
      number /= 2
if __name__ == "__main__":
   main()
```

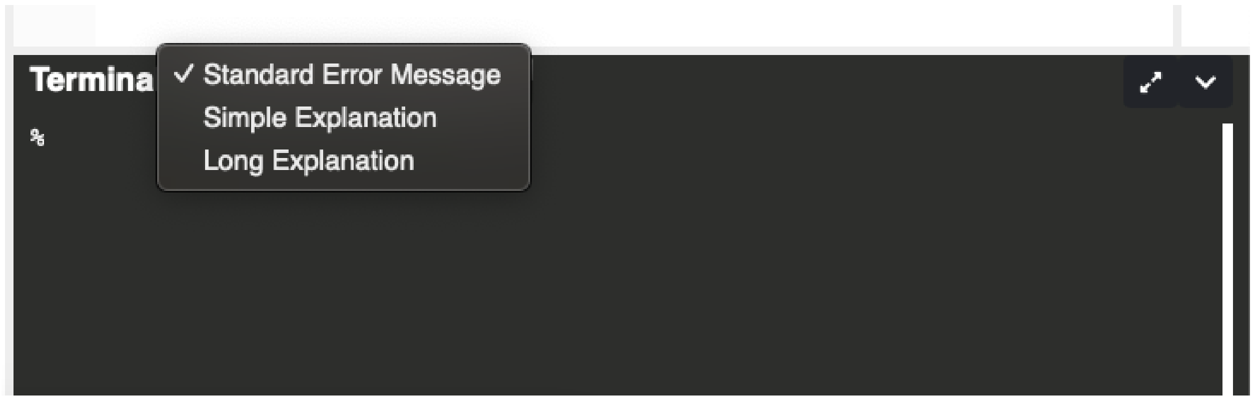
If you notice that a line of code is running too frequently or infrequently in a program with nested loops, you should check which loop the code is in. Is it in the outer loop but outside the inner loop or inside both the inner and the outer loop?

General Tips for Debugging

Programming is typically about 50% writing code and 50% trying to fix code that you've already written, or debugging. Debugging can take anywhere from a few seconds to hours to fix, depending on the nature of the bug and what tricks you use to find it. Here are a few quick methods for cutting down your debugging time:

1. Make sure you understand the error.

Oftentimes, when coding in the real world, programmers can get errors with descriptions that seem to have little to do with the actual bug. Fortunately, in the IDE, you have options to make errors easier to understand.



2. Trace through the error's origin.

When you get an error message, the message will often have the names of the files and functions where the errors occurred. If the error occurred in a function that was called by another function, you will see that reflected in the error output:

```
def error_function():
   x = 5 + '7'
def main():
   error_function()
```

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

If you run this program, you may see these 3 lines:

```
File "<exec>", line 13, in mainApp
File "<exec>", line 10, in main
File "<exec>", line 7, in error_function
```

This is the trace of where the errors occur in our program. The bottom line is the function where the error actually occurs. This is **so** useful when debugging. An error message can tell you what went wrong AND where to find the buggy code.

3. Use print statements

Print statements can be very useful when debugging. Consider this example:

```
def main():
   x = 51 % 18 * 12 - 9 / (4 ** -2)
   y = x + (-(3 * x)) - 14
   z = -y - x / 6 + (-(8 + x * 2))
   w = z / (4 + 1) * 2 % 7

   print('x: ' + str(x))
   print('y: ' + str(y))
   print('z: ' + str(z))
   print('w: ' + str(w))

   print((x + y + z + w) / (x * y * z * w))
if __name__ == '__main__':
   main()
```

Let's walk through this error. The error is: `ZeroDivisionError: float division by zero`. Look at the denominator of the print statement: `x * y * z * w`. One of these variables must have been assigned `0`. We could go through these four lines:

```
x = 51 % 18 * 12 - 9 / (4 ** -2)
y = x + (-(3 * x)) - 14
z = -y - x / 6 + (-(8 + x * 2))
w = z / (4 + 1) * 2 % 7
```

but these statements are unnecessarily complex and evaluating them could take longer than we would like. The print statements that we added make it much easier to see what each variable's value is. After running the code, scroll to the top of the error message. You'll see our print messages at the top which show that `z` and `w` both equal `0`. Now, we can focus our energy on fixing `z` and `w`.

Print statements are also useful for seeing how a problematic variable changes within a loop or if a line of code is even reached (like code within an if statement).

4. When debugging a program with the random library, use `random.seed()`.

Sometimes a program breaks only on certain conditions. When using the random library, sometimes, certain numbers cause errors while others don't. Using `random.seed` can help. Using different seeds, you can determine exactly which numbers are causing the issue, so that it can be fixed.

5. Test functions individually.

This debugging tip should come *before* you reach a bug. It should happen while you are writing your code. Be sure to test each helper function individually and as you are writing them (i.e. before you finish the whole program). This will allow you to catch bugs early, and it will make them so much easier to find. Looking through one function for a bug is much easier than sifting through an entire program.

Style Guide

Introduction to Programming Style in Python

Created by Juliette Woodrow

As you start to write your first programs, it's important to understand the concept of programming style and why it is essential in the development process. In this handout, we will discuss the importance of good programming style and provide guidelines to help you write clean and maintainable code.

Why is Programming Style Important?

Good programming style is essential for several reasons:

1. It makes your code easier to read and understand.
2. It helps you and others identify and fix bugs more quickly.
3. It enables better collaboration with other programmers.
4. It leads to more maintainable and reusable code.

"Readability" is the key factor to consider when it comes to programming style. Code readability is crucial for understanding, maintaining, and collaborating on a coding project. Writing readable code makes it easier for you and others to identify and fix bugs, build upon existing code, and work together on projects. In this handout we will discuss many strategies for writing code that is easy to read and understand.

Now, let's dive into some specific programming style guidelines for Python:

Descriptive Variable and Function Names

Choosing descriptive variable and function names is an essential aspect of writing clean and maintainable code. Meaningful names make it easier for you and others to understand the purpose of each component in your code, reducing the need for extensive comments and making your code more self-explanatory. In Python, we follow the snake_case naming convention for variables and functions. Here are some guidelines to help you choose the right names:

1. Descriptive names: Select clear and descriptive names for your variables and functions. This makes it easier for you and others to quickly grasp the functionality of your program.

Example:

```
# Good
def move_to_wall():
    .....

# Bad
def karel_moving():
    .....
```

Karel can move in a lot of ways and so we need to be more specific and descriptive when choosing the name of the function so that it is easier to understand exactly what the function does.

2. Snake case: In Python, use snake_case for variable and function names. This means using lowercase letters and separating words with underscores. This naming convention is easy to read and is the standard practice in Python.

Example:

```
# Good
def calculate_total_price(price, tax_rate):
    total_price = price * (1 + tax_rate)
    return total_price

# Bad
def CalculateTotalPrice(p, tr):
    x = p * (1 + tr)
    return x
```

In the second example, it is harder to understand what the function does. Though the person writing the function likely knows that p means price and tr means tax rate, these names are not clear to other people reading the code (and the author if they look back at this code later). Though the function name is clear, it is not written in snake case. In python programs, it is standard to use snake case for all function names.

3. Avoid Python Keywords as Names
   In addition to picking descriptive names, there are certain python keywords that we do not want to use when naming our own variables and functions as that can be confusing. Python keywords that should not be used as function or variable names: `False`, `True`, `None`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

By following these guidelines and using descriptive variable and function names written in snake_case, you can greatly improve the readability and maintainability of your Python code.

Using Whitespace to Improve Code Readability

Whitespace helps visually separate different sections of your code and makes it easier to understand the structure and flow of your program. To effectively use whitespace for better code readability, follow these guidelines:

1. Use consistent indentation to show the hierarchy of your code blocks, typically 4 spaces per indentation level in Python. Luckily, a tab is typically 4 spaces. So you can just hit the tab button instead of typing out 4 spaces each time.
2. Separate function definitions with two blank lines to clearly distinguish between them.
3. Add a single blank line between logical sections within a function to group related lines of code together.
4. Surround operators with spaces to make expressions easier to read. Instead of `x=3+4` you should write `x = 3 + 4`.

By following these whitespace guidelines, you can make your code more visually appealing and easier to navigate. This may seem time consuming, but it will save you lots of time in your editing and debugging process.

Comments

Comments help you and others understand the purpose and functionality of your code, making it easier to debug, maintain, and collaborate on projects. However, it's also essential to strike the right balance between commenting and not over-commenting. There are two standard types of comments in python. In-line comments are comments that start with a # and go only to the end of the line. Docstring comments are comments inside a set of three triple quotes and can be more than one line. We discuss below when to use each type of comment. Here are some guidelines for using comments effectively in your code:

1. Comment for each function: Write a short comment for each function in your program to explain its purpose and how it works. This helps others quickly understand the function's role and functionality within the code. Comments should go below the def and before the code body to make it clear which function this comment belongs to. It is standard in python for comments at the top of each function to be inside of a doc string as shown below.

Example:

```python
def calculate_area(radius):
    """Calculate the area of a circle given its radius."""
    area = 3.14 * radius ** 2
    return area
```

2. Comment at the top of each file: Include a comment at the beginning of each file in your program, describing its overall purpose and any important information about its contents. This helps provide context for the code in the file and makes it easier for others to understand how it fits into the larger project. It is standard in Python to place file header comments inside a doc string

Example:

```python
"""
Filename: geometry.py
Author: Juliette
Description: This module contains functions for calculating the area,
perimeter, and volume of various geometric shapes.
"""
...
your_code_here
...
```

3. Inline comments for complex code: Use inline comments to provide explanations for any code that might be difficult to understand or requires additional clarification. Avoid writing comments for straightforward code, as this can clutter your code and make it harder to read.

Example:

```python
# Good
def calculate_discount(price, discount_rate):
    if discount_rate <= 0.1:
        discount = price * discount_rate
    elif discount_rate <= 0.2:
        # Apply an additional 2% discount for rates between 10% and 20%
        discount = price * (discount_rate + 0.02)
    else:
        discount = price * 0.25  # Maximum discount of 25%

    return discount
```

```
# Some redundant / unnecessary comments
def calculate_discount(price, discount_rate):
    if discount_rate <= 0.1:  # Check if discount rate is less than or equal to 10%
        discount = price * discount_rate  # Calculate discount
    elif discount_rate <= 0.2:  # Check if discount rate is less than or equal to 20%
        discount = price * (discount_rate + 0.02)  # Calculate discount with additional 2%
    else:  # Discount rate is greater than 20%
        discount = price * 0.25  # Calculate discount with maximum 25% rate

    return discount
```

In the “Good” example, the in-line comments are used to provide clear explanations for any non-obvious or potentially confusing parts of the code, such as the additional 2% discount applied for rates between 10% and 20%, and the maximum discount of 25%.

In the “redundant / unnecessary comments” example, the in-line comments are overly wordy and explain parts of the code that are already clear enough. This may distract the reader and make the code harder to understand.

Decomposition

Decomposition is the process of breaking down a complex problem into smaller, more manageable parts. In programming, this often means dividing your code into smaller, reusable functions. This helps make your code easier to understand, debug, and maintain. Let’s discuss some guidelines to follow when decomposing your code:

1. Do one thing: Each function should have a single responsibility, meaning it should only do one conceptual thing, like building a hospital, not building a hospital and then moving to the next hospital. This makes your code easier to reuse and modify in the future. It will also make the debugging process easier because you will be able to narrow the issue down to a single function.

2. Know what it does by looking at its name: Choose function names that clearly describe their purpose, so you can understand what they do just by looking at their names. This improves code readability and reduces the need for extensive comments.

3. Less than 10 lines, 3 levels of indentation: Keep your functions short (preferably

less than 10 lines) and limit the levels of indentation to 3 or fewer. Note that 10 lines / 3 levels of indentation is not a strict rule. Rather these are guidelines to keep in mind as you are starting out in your programming journey. This helps keep your code readable and maintainable, as it's easier to understand the logic and flow of shorter functions.

4. Reusable and easy to modify: Write code that is easy to reuse and modify by using functions. This allows you to save time and effort when making changes or building upon your existing codebase.

Constants and Magic Numbers

In programming, it's important to understand the concept of constants and magic numbers, as well as how to use them properly. This section will explain what constants and magic numbers are and provide guidelines for using them effectively in your code.

Constants

Constants are values that do not change during the execution of a program. They are used to give meaningful names to certain values, making the code easier to understand. In Python, constants are typically defined in upper snake case, which uses uppercase letters with words separated by underscores.

Example:

```
PI = 3.14159
GRAVITY = 9.81
MAX_CONNECTIONS = 100
```

By using constants, you can easily update the value in one place if it needs to change, without having to search for and modify multiple occurrences of the same value throughout your code.

Magic Numbers

Magic numbers are hardcoded values that appear directly in your code without any explanation of their meaning or purpose. These values can make your code difficult to understand and maintain, as it's not clear where they come from or why they are being used.

Example:

```python
# Good
PI = 3.14159
area = PI * radius ** 2

# Bad
area = 3.14159 * radius ** 2
```

To avoid magic numbers in your code, replace magic numbers with named constants, as shown in the example above.

Parameters

Parameters are pieces of information that functions need to use when we call them. You can think of them as inputs to the function. By using parameters, functions can work with different inputs without changing the code inside the function. This makes the function more adaptable and easy to use in various situations. Parameters help functions focus on the information they receive as input and not depend on information from other parts of the program. Using parameters is another example of good programming style.

I think it is helpful to understand how parameters can improve programming style through an example. Let's consider an example where we want to create a function that adds two numbers together. First, let's look at a version without using parameters:

```python
# Without parameters
def add_numbers():
    a = 5
    b = 3
    result = a + b
    print(result)

add_numbers()  # Prints out: 8

# If we want to add different numbers, we need to change the values of 'a' and
'b' inside the function
def add_numbers():
    a = 7
    b = 4
    result = a + b
    print(result)
```

```
add_numbers()  # Prints out: 11
```

In the above example, we have to change the values of `a` and `b` inside the function every time we want to add different numbers. Now, let’s rewrite the function using parameters:

```
# With parameters
def add_numbers(a, b):
    result = a + b
    print(result)

add_numbers(5, 3)  # Prints out: 8
add_numbers(7, 4)  # Prints out: 11
```

By using parameters, we can now call the `add_numbers` function with different inputs without changing the code inside the function. This makes the function more flexible and reusable, allowing us to easily add different pairs of numbers without modifying the function itself.

Advantages of using parameters:

1. Modularity: Functions with parameters work independently and can be used in different parts of your program or even in other projects without any problems.
2. Ease of maintenance: When you need to change a function’s behavior, you’ll only have to change the code inside the function. This won’t affect the rest of your program.
3. Simpler testing: It’s easier to test functions that use parameters because you can provide different inputs to check how the function behaves in various scenarios without changing the function itself.

Conclusion

Remember that your code is not only meant for computers to execute but also for humans to read and understand. By consistently practicing good programming style, you will develop strong coding habits that will benefit you and your collaborators in the long run, making you a more effective and efficient programmer.

This guide is a rundown of some of the most important, official Python rules that we want to emphasize while you are first learning how to program. As mentioned above, not all of these are strict rules so this should be used as guidance and inspiration. Some style tactics are up to programmer preference, and in that case, readability should be what you’re going for. Feel free to ask your section leader or post on the forum if you’re unsure about style rules.

If you want even more style advice, you can check out the official python style guide. This. has more advanced topics that are outside the scope of this course and not necessarily things we emphasize to people who are just learning how to program, but feel free to peruse at your leisure.

Happy coding!

— Containers —

Lists

Quest

When we store a piece of data for our code to use later, we assign that value to a variable. Sometimes, however, when there is a lot of related information to keep track of, it can get messy trying to make a bunch of variables for each singular piece of data. You're going to learn a couple of better ways to store multiple values inside a single variable, and the first is known as a list.

Can you do me a favor? I'm organizing a marathon, and I need you to write down how long the winner took to finish the race and store that as a variable. Actually, I need you to time the first 10 finishers, so you'll have to create ten variables (one for each runner). *Actually...* I want you to keep track of *all* the runners. There are 2000 of them! You can imagine how crazy it would be to define and keep track of 2000 variables! We're going to need a better system for efficiently storing large amounts of data like this. Lucky for you, I've got just the thing!

Lists

You've probably written a list down before. Perhaps it was a list of things to do or a list of groceries to buy from the store. Lists are just a collection of multiple pieces of information stored in a particular order. In Python, lists are no different! You can think about a list in Python as a table of values. Each value is labeled sequentially to give you a sense of how the data is ordered. These values start with the integer 0 and count up. Below is an example of a list of names!

“Karel”	“Justin”	“Ki”	“Chris”	“April”	“Mehran”	“Elyse”	“Hannah”
0	1	2	3	4	5	6	7

The number below each name represents that item's index or placement in the list. The item at index 0 is the first element of the list, followed by the item at index 1, followed by 2, and so on.

Wait? Why do we start with a 0? Wouldn't it make more sense to start with 1 as the first index???

There are several reasons for this, and it has to do with some mathematics and the way computers are designed. It'll take some getting used to, but Python is a 0-indexed language, meaning anytime something in Python involves an index, the first element is going to start at index 0. This means the index of every item in the list is one less than its actual place in the list. It's confusing, we know, but it's also important that you realize this now before we start using lists in our programs.

A New Type of Data

A list is an example of a new type of data we haven't seen before called a container. We call it that because... well... it contains things. While each container is a new type, the job of the container is to store one or more other pieces of data. A `list` might have an `int`, `float`, `str`, or even another `list` inside of it.

Mutability

Primitive types, like `int`, `float`, and `str`, are immutable, meaning they cannot be edited or mutated.

What exactly do we mean by *edited*?

To demonstrate this, let's assign the integer `6` to a variable

```
my_favorite_number = 6
```

Now, I want you to *edit* that 6 in some way. Maybe you suggest that we double it. Let's do that!

```
my_favorite_number = 6

# Now my_favorite number is 12
my_favorite_number *= 2
```

Boom! We just edited `6` to make it `12`... right? Not exactly.

6 is an integer, and so is `12`. Both are separate, immutable pieces of data. In the code above, we took the integer `6` and assigned it the name `my_favorite_number`. When we multiplied `my_favorite_number` by 2, here's what *really* happened. Instead of actually doing anything to the integer `6`, Python just grabbed the integer `12` and assigned *that* to `my_favorite_number` instead. You can't edit a `6`, it's just a `6`. When we change the value of a variable that is an immutable type, we are actually just assigning that variable to an entirely new piece of data.

Now we can talk about lists, which *are* mutable. As you will soon see, lists are mutable because you can actually edit what is inside them without creating a whole new list.

Defining Lists

How do we actually create one of these fancy lists in Python? There are many ways to do it, and below are examples of a few common ones:

Defining a list of known values:

```
# Individual elements are seperated by commas
name_of_list = [item1, item2, item3]
```

Defining a list of a certain size with default values

```
# This creates a list where every element is equal to initial_value. The number of
# elements in that list is equal to size
name_of_list = [initial_value] * size
```

Defining an empty list:

```
name_of_list = []
```

Here, the square brackets `[]` are used to signify that the data we are working with is a list. These brackets appear when we try to print a list as well!

```
'''
this program stores the names of several dogs in a list and then prints that
list to the console
'''
def main():
   my_dogs = ["Tyler", "Paisley", "Charlie", "Cooper", "Sam"]
   print(my_dogs)
if __name__ == '__main__':
   main()
```

Accessing List Elements

So, we've made a list and it has all these values in it. How do we get them out? When you want to view an element inside a list, you look that value up using its index! Recall the list of names we wrote out above:

“Karel”	“Justin”	“Ki”	“Chris”	“April”	“Mehran”	“Elyse”	“Hannah”
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Say we wanted to get the name `"Chris"` from this list. To do so, you would ask the list for the element stored at index `3`, and the list would return the string `"Chris"`! To do this in Python, we use those brackets `[]` again, but in a slightly different way.

The template for accessing an item in a list looks something like this:

```
name_of_list[index_of_element]
```

The brackets are attached to the end of the name of the list, indicating to the computer that we are attempting to access an element of that particular list. Between the brackets, we put the index of whichever element we want the computer to go and get for us.

Let's try to get the name `"Chris"` out of our list of names, but this time we're going to do it in Python!

```
# Here is our list of names
names = ["Karel", "Justin", "Ki", "Chris", "April", "Mehran", "Elyse",
"Hannah"]
# I want the name "Chris" which is at index 3
current_name = names[3]
# Let's print this to make sure we actually got the right value
print(current_name)
```

It is easy to forget that each element's index is one less than its place in the list, so always double-check when writing your code that you are accessing the correct element. Also, if you try to access an element at an index that is *not* in the list, you will get an 🛑 **Error**!

```
# Here is our list of names
names = ["Karel", "Justin", "Ki", "Chris", "April", "Mehran", "Elyse",
"Hannah"]
# The highest index of this list is 7, but let's ask for the element at index
# 8
# What could possibly go wrong?
current_name = names[8]
# We'll never get this far :(
print(current_name)
```

Editing a List

When we use the bracket notation to access an element of a list, we are actually peering inside the list and getting direct access to that data. We can do more than just read values from a list, we can edit those values too! This right here is what makes lists mutable. If we want a list to change, we don't have to create a whole new list. We can just go inside the list we already have and change some of the information it contains.

When you want to assign a new value to a variable, you write that variable's name followed by an equals sign and the new value. To edit an item in a list, we do the same thing with just with bracket notation:

```
# Here is a list of my three favorite fruits
favorite_fruits = ["Strawberries", "Blueberries", "Apples"]
# Ooh, but I just had my first papaya and I LOVED it!!! I think I liked it more
# than I like Apples, so I want to replace "Apples" with "Papayas" in my list
# Apples is at index 2, so we want to edit favorite_fruits[2]
favorite_fruits[2] = "Papayas"
# Did it work?
print(favorite_fruits)
```

Slices

Sometimes you need access to a portion of an entire list. Python allows you to request a sublist, also known as a **slice** of a list, using the square brackets.

To take a slice of a list, use the square brackets in the following way:

```
name_of_list[beginning:end]
```

This code means, "Give me all the elements between the index `beginning` and the index `end`." *However, be careful!* The element at index `beginning` is included in the slice, but the element at index `end` is *not!* The slice will contain every element from `beginning` up to but not including `end`. If, as an example, I want my slice to include the item at index `3`, then my slice should go to index `4` so that `3` is included.

Let's see this in action:

```
# We start with a list with a total of 5 elements
# The first index is 0 and the last index is 4
letters = ["A","B", "C", "D", "E"]
# We want the elements "B", "C", and "D"
# "B" is at index 1 and "D" is at index 3
# So we get the slice from index 1 to index 4
print(letters[1:4])
```

Say you wanted a slice that starts at index 0. If you remove `beginning` from the brackets but leave the colon and `end`, Python will start the slice at 0 by default.

```
name_of_list[:end]
```

Check this out below:

```
# We start with a list with a total of 5 elements
# The first index is 0 and the last index is 4
letters = ["A","B", "C", "D", "E"]
# We want the first 3 elements "A", "B", and "C"
# "A" is at index 0 and "C" is at index 2
# So we get the slice from index 0 to index 3
# [:3] means everything from index 0 up to but not including 3
print(letters[:3])
```

The same goes for a slice that ends at the final index of the list. If you remove the `end` from the brackets but leave the `beginning` and the colon, Python will end the slice at the last item of the list.

```
name_of_list[beginning:]
```

As an example:

```
# We start with a list with a total of 5 elements
# The first index is 0 and the last index is 4
letters = ["A","B", "C", "D", "E"]
# We want the last 2 elements "D", and "E"
# "D" is at index 3 and "E" is at index 4 (the end of the list)
# So we get the slice from index 3 to index 5
# [3:] means everything from index 3 all the way to the end of the list
print(letters[3:])
```

If we didn't use this shorthand, getting the last two elements from the list above would require taking a slice from index `3` to index `5`. This may seem strange because the list only goes up to index `4`. Part of the benefit of using the shorthand is that you don't need to think about this. Since, technically, `end` is not included in the slice, `end` needs to be at the index one more than the last index if you want the slice to include the last element.

Adding and Removing Elements

To add an element to the end of a list, we use the `append` function:

```
colors = ["Red", "Yellow", "Orange"]
colors.append("Green")
print(colors)
```

This is a special function that is a part of the list itself. This is why we have to connect the name of the function to the list with a period. You saw these in the graphics section, where functions were attached to the `canvas` variable because they are acting on that particular canvas we create. You'll understand more about these functions later.

To remove an element, we use the `remove` function:

```
colors = ["Red", "Yellow", "Orange"]
colors.remove("Red")
print(colors)
```

If you try to remove an element that appears multiple times in the same list, Python will only remove the first instance of that element:

```
colors = ["Red", "Yellow", "Red"]
colors.remove("Red")
print(colors)
```

Finding Elements in a List

There are a couple of ways to search a list for a particular item!

To check whether an item is contained in a given list, we use the keyword `in`:

```
colors = ["Red", "Yellow", "Orange"]
my_color = "Red"
# This logical expression reads "my_color is an element in the list colors"
if my_color in colors:
   print(my_color, "is in the list!")
else:
   print(my_color, "is not in the list!")
```

To find the index of a specific element in the list, we use the `index` function:

```
colors = ["Red", "Yellow", "Orange"]
my_color = "Red"
red_index = colors.index(my_color)
print("Red is at index", red_index)
```

Two important things to note about the `index` function:

1. If the item you are looking for appears more than once in the list, the index returned is the index of the first occurrence of that element.

```
colors = ["Red", "Red", "Red"]
my_color = "Red"
red_index = colors.index(my_color)
print("Red is at index", red_index)
```

2. If the item you are looking for is *not* in the list, the program will crash and you will get an 🛑 **Error**!

```
colors = ["Red", "Yellow", "Orange"]
my_color = "Blue"
red_index = colors.index(my_color)
print("Red is at index", red_index)
```

Size of a List

To get the number of elements contained in a list, we use the `len` function. Unlike functions such as `append` and `index`, which are attached to the name of the list with a period, `len` is a function that takes the list in an argument:

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

```
size_of_list = len(colors)
print("The list has", size_of_list, "elements in it.")
```

Looping over a List

If you just print a list, you'll see all of its elements in the console:

```
integers = [10, 4, 1, 3, 2]
print(integers)
```

This doesn't actually let you do anything with the data in the list, it just prints it out. What if we wanted our code to do something for every element of the list?

We can access each element of a list using a for-loop. To do this, we loop over a range up to the size of the list. In each iteration of the loop, we access the element at index `i` (the iterator from the for loop).

```
# We have a list of positive integers
integers = [10,4, 1, 3, 2]
size_of_list = len(integers)
for i in range(size_of_list):
   print(integers[i])
```

Lists Can Store Anything

Seriously, you can put data of any type inside a list. They don't even need to all be the same type!

```
# This is a perfectly valid list
mystery_box = [17, "A rabbit", False, 22.1]
```

Usually, we'll stick to one type of data per list, just so it's easier to work with.

Nested Lists

I love lists so much! I have a list for everything: my favorite foods, colors, animals, you name it! What if... hear me out... what if we made a *list of all my lists!!!!!* Lists can even store other lists. These are called **nested lists** because one list sits inside of the other:

```
# Here are my favorite colors
colors = ["Cyan", "Purple", "Yellow"]
# These are my favorite foods
foods = ["Chicken", "Soup", "Rice"]
# And these are my favorite animals
animals = ["Pandas", "Lions", "Elephants"]
# Now, I'm going to make a list that contains all my lists
favorites = [colors, foods, animals]
```

```
# What do you think this will do?
print(favorites[0])
```

At each index of `favorites`, there is another list. Each of these lists has its own elements. If you wanted to get the entire list of colors, you would access the index of `favorites` where `colors` is placed. That's why the code above prints the list of colors. `colors` is the item at index `0` in `favorites`.

If you want a specific item within a nested list, you have to two sets of square brackets.

```
item = outer_list[index_of_inner_list][index_of_item_in_inner_list]
```

The first set of brackets tells Python which list you want to open, and the second set of brackets tells you which index from that inner list to pull from:

```
# Let's define the big favorites list again
colors = ["Cyan", "Purple", "Yellow"]
foods = ["Chicken", "Soup", "Rice"]
animals = ["Pandas", "Lions", "Elephants"]
favorites = [colors, foods, animals]
# Now, I want to pull "Soup" out from the foods list!
# foods is at index 1 in favorites and "Soup" is at index 1 in foods
# So this is how we write that in code
print(favorites[1][1])
```

Negative Indices

Using a negative-valued index when accessing elements of a list wraps around to the end of the list:

```
# Let's define a list of colors
colors = ["Red", "Yellow", "Orange", "Green", "Purple", "Blue"]
# I want the second to last item in the list. There are a few ways to get it
# Option 1: We just know "Purple" is at index 4
elem_from_index = colors[4]
# Option 2:  We can calculate the second to last index using len
last_index = len(colors) - 1 # Remember to subtract 1 since lists are 0-index
elem_from_len = colors[last_index-1]
# Option 3: We use a negative index. [-2] means "The second to last element"
elem_from_neg_index = colors[-2]
print("Element Retrieved Using Index:", elem_from_index)
print("Element Calculated From Size of List:", elem_from_len)
print("Element Retrieved Using Negative Index:", elem_from_neg_index)
```

You can also use negative indices to get a slice of a list:

```
# Let's define a list of colors
colors = ["Red", "Yellow", "Orange", "Green", "Purple", "Blue"]
# I want the "Orange", "Green", and 3 items of the list. There are a few ways
to
    get them
# Option 1: We just know that the last three items start at index 3
slice_from_index = colors[3:]
# Option 2: We can calculate the third to last index using len
last_index = len(colors) - 1 # Remember to subtract 1 since lists are 0-index
slice_from_len = colors[last_index-2:]
# Option 3: We use a negative index. [-3:] means "Take a slice starting at the
# third to last element"
slice_from_neg_index = colors[-3:]
print("Slice Retrieved Using Positive Index", slice_from_index)
print("Slice Retrieved From Using Size of List", slice_from_len)
print("Slice Retrieved Using Negative Index", slice_from_neg_index)
```

For-Each Loops

Quest

For loops are a great tool for repeating some code multiple times. When it comes to containers, it is often that we want to write code that repeats *for each* item in the container. Python gives us a simple way to do that!

For-Each Loops

If you have a container like a list or a dictionary, we use a for-each loop to repeat once for every item in the container. Here's how that looks in Python:

```
for element in container:
    # Each time we loop, element becomes the next item in the list
```

Iterating over Lists

For-each loops work differently on different containers. With lists, the iterator `element` loops over all the values in the list. You can change the name of `element` to be whatever you like, just as long as the name hasn't been used yet:

```
colors = ["Red", "Yellow", "Blue"]
# Instead of printing the whole list, we print each individual element
for color in colors:
   print(color)
```

Iterating over Dictionaries:

With dictionaries, a for-each loop will iterate over all the keys of the dictionary:

```
grocery_list = {
   'lemons': 3,
   'apple juice': 1,
   'honey': 2
}
# Instead of printing the whole dictionary, we print each key individually
for item in grocery_list:
   print(item)
```

If you want to loop over all the values of a dictionary, you can either access them in each loop or you can loop over `dict.values()`

```python
grocery_list = {
   'lemons': 3,
   'apple juice': 1,
   'honey': 2
}
# Option 1: Access each value individually using bracket notation
print("Bracket Notation:")
for item in grocery_list:
   print(grocery_list[item])

# Option 2: Access each value individually from grocery_list.values()
print("grocery_list.values()")
for item in grocery_list.values():
   print(item)
```

Fun Fact

Before, when we were just looping over some code a fixed number of times, we would use

`for i in range(...)` with the number of loops written inside the parentheses. Fun fact, this is technically also a for-each loop because `range` returns a sequence of numbers to be iterated over.

Dictionaries

Quest

Congratulations on learning your first container! Lists are a very useful tool in Python, and you will likely use them often. Building on our momentum from the last two sections, we are going to go right into your second container: dictionaries!

When you see the word dictionary you might think of something like this:

Code in Place Dictionary

1. animation loop (noun) – a while loop used to repeatedly update a graphics window or canvas to produce the appearance of movement

2. nested (adjective) – structures or statements arranged in a hierarchy (one within another)
3. return (verb) –
    1. to end a function
    2. to give back a value (called the return-value) in the context of a function

Real life dictionaries are essentially a group of words and their corresponding definitions. In Python, dictionaries refer to key value pairs where:

- the key is some unique identifier
- the value is something we associate with that key

An example you might be familiar with would be the list of contacts on your phone or in your phonebook. The keys would be the names of the people and the values would be the corresponding phone numbers. So how do we create dictionaries in Python? Let's look at the syntax with a few examples below:

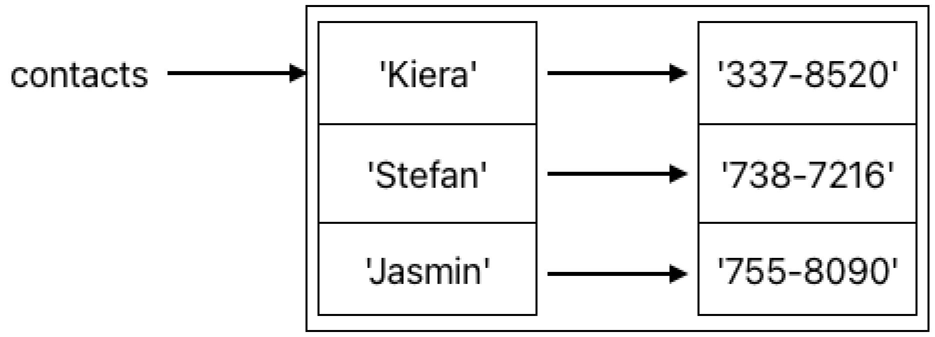
```
contacts = {'Kiera': '337-8520', 'Stefan': '738-7216', 'Jasmin': '755-8090'}
```

As you can see, there are a few things to note about the syntax of dictionaries:

1. Dictionaries start and end with curly braces `{ }`
2. Between each key and the corresponding value is a colon `:`

3. Each pair is separated by a comma `,`

Conceptually, the image below is essentially what the above line of code does. `contacts` refers to the container as a whole where each key is mapped to the corresponding value:



Keys and values don't have to be strings. So, what can they be?

Keys must be immutable types. We briefly explained what we mean by immutable in the last section (types that can't be edited). These are things like ints, floats, strings and bools.

Values can be mutable or immutable. For example, you can have lists as values or even other dictionaries as values. Keep reading to see an example of a nested dictionary.

Building up a Dictionary

If we don't know what we want to put in a dictionary when we declare it, we can start with an empty dictionary.

```
empty_dict = {}
```

If we want to add a value, we can do so like this:

```
final_grades = {}
final_grades['Viktor'] = 90
```

We use square brackets `[]` to access things in the dictionary just like we do with lists. We can grab values using keys instead of indexes. Unlike with lists, referencing a key that has not yet been added to the dictionary will not error, but add the key value pair.

We can even update the value in a key value pair in the same way that you would a normal variable.

```python
def main():
    # build the car1 dict
    car_info = {}
    car_info['make'] = 'honda'
    car_info['model'] = 'civic'
    car_info['year'] = 2016
    car_info['miles'] = 3000
    car_info['crashed?'] = False

    # print the current dictionary
    print(car_info)

    # update values based on new information
    car_info['crashed?'] = True     # car1 gets into a crash
    car_info['miles'] += 20         # car1 drives 20 more miles

    # prints the updated values
    print(car_info['miles'])
    print(car_info['crashed?'])

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

Reassigning a value to an existing key does not create a new key value pair. It replaces the old value in the existing pair with the new value. For example, the line `car1_info['crashed?'] = True` replaces the value `False` inside of `car1_info` with `True`. `car1_info` still only has one key called `'crashed?'`. On the other hand, values are allowed to have duplicates.

Another thing to note about keys: if you try to access a key that doesn't exist in the current directory, you will get an error 🛑:

```python
def main():
    # fill final grades dict
    final_grades = {}
    final_grades['Viktor'] = 88
    final_grades['Paola'] = 92
    final_grades['Ella'] = 92
    final_grades['Yasser'] = 97

    # what if we forgot to add someone's grade to the dict?
    print(final_grades['Cyrus'])

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

To prevent this error, we can check to see if a key is in the dictionary before we try and access its value using the `in` keyword.

```python
def main():
    # fill final grades dict
    final_grades = {}
    final_grades['Viktor'] = 88
    final_grades['Paola'] = 92
    final_grades['Ella'] = 92
    final_grades['Yasser'] = 97

    # what if we forgot to add someone's grade to the dict?
    if 'Cyrus' in final_grades:
        print("Cyrus's grade: " + final_grades['Cyrus'])

    if 'Ella' in final_grades:
        print("Ella's grade: " + str(final_grades['Ella']))
if __name__ == '__main__':
    main()
```

Nested Dictionaries

Earlier in this section, we mentioned that it is possible to create nested dictionaries in Python. But what might this look like? Let's say we wanted to implement a contacts app for phones (like our contacts dictionary from earlier). In addition to phone numbers, we also want each contact to be able to store other information like emails, birthdays, etc. We could do so like this:

```python
def main():
    # you can build nested contacts dict directly...
    contacts = {'Kiera': {'number': '337-8520', 'birthday': 'March 2nd',
                    'email': 'kgomez4@gmail.com'}}

    # or add key value pairs later just like with regular values
    contacts['Stefan'] = {'number': '738-7216', 'birthday': 'July 24'}
    contacts['Jasmin'] = {'number': '755-8090', 'email': 'jas.shah@gmail.com'}
    # to access the inner dictionary elements, use second set of brackets
    contacts['Jasmin']['birthday'] = contacts['Stefan']['birthday']

    print(contacts)


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

Mutability

As we said before, nested dictionaries like the one above are only possible because values can be mutable. Dictionaries and lists are both examples of mutable types. In the last section, we described this as a new kind of variable type, which unlike immutables, could be changed or altered (for both lists and dictionaries, this is accomplished using bracket notation). But is that all that mutability means? It turns out that mutability also affects arguments to functions. In the section on scope, we showed you that if you passed a variable as an argument to a helper function, that variable would remain unchanged in the original function (without returning and reassignment). What we didn't tell you, is that this is only true for immutable variable types. Mutable types like lists and dictionaries will change if you pass them into a function.

Consider the following example using our contacts app. Now that we've updated the contacts app to keep track of more information for each contact, we want to allow users to add new contacts. We could use a helper function for that:

```python
def add_contact(contacts, name, number, birthday=None, email=None):
    '''
    builds a contact based on the given info and then adds it to contacts
    params:
        name (str): name of the contact
        number (str): phone number of the contact
        birthday (str or None): birthday of the contact (optional)
        email (str or None): email of the contact (optional)
    '''
    contact = {'number': number}

    # birthday and email are 'optional' arguments because they have
    # default values, but we don't want None in our contacts dict
    if birthday != None :
        contact['birthday'] = birthday

    if email != None :
        contact['email'] = email

    contacts[name] = contact
    # notice how we don't return contact
def main():
    contacts = {}

    # add the three contacts using our helper function
    add_contact(contacts, 'Kiera', '337-8520',
              'March 2nd', 'kgomez4@gmail.com')
    add_contact(contacts, 'Stefan', '738-7216', 'July 24')
    add_contact(contacts, 'Jasmin', '755-8090',
              email = 'jas.shah@gmail.com')

    print(contacts)


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

Even though contacts isn't returned, the helper function still updates contacts in main. Mutability is useful because it allows us to edit dictionaries and lists without having to return them and reassign them every time, but it can also be tricky. Sometimes, you might want to write a function that takes in a dictionary as a parameter in order to read from it, but not alter it. If you aren't careful, you could end up with a bug in your code that changes the dictionary in the helper function thereby changing the original dictionary from the caller function.

It is not just arguments that mutability affects either. Look at the code below:

```
def add_contact(contacts, name, number, birthday=None, email=None):
    '''
    builds a contact based on the given info and then adds it to contacts
    params:
        name (str): name of the contact
        number (str): phone number of the contact
        birthday (str or None): birthday of the contact (optional)
        email (str or None): email of the contact (optional)
    '''
    contact = {'number': number}

    if birthday != None :
        contact['birthday'] = birthday

    if email != None :
        contact['email'] = email

    contacts[name] = contact
def main():
    contacts = {}

    add_contact(contacts, 'Kiera', '337-8520',
             'March 2nd', 'kgomez4@gmail.com')
    add_contact(contacts, 'Stefan', '738-7216', 'July 24')
    add_contact(contacts, 'Jasmin', '755-8090',
             email = 'jas.shah@gmail.com')

    # assign 'Jasmin' dict to a new variable
    jasmin = contacts['Jasmin']

    # change new variable
    jasmin['email'] = 'new_email@gmail.com'

    # contacts will be affected as well
    print(contacts['Jasmin'])


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

As you can see, changing `jasmin` also changed `contacts`. This is different from the behavior we saw from variables in basic arithmetic. Any assignment reference to a dictionary or list (or other mutable type) will exhibit this behavior. If you are curious as to why this occurs, we will take a deeper dive into mutability in a later section.

Useful Functions with Dictionaries

Let's look at some useful functions for dealing with dicts.

Deleting Key Value Pairs

```
def main():
    squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
    # dict.pop(key) deletes the key-value pair and returns value
    deleted = squares.pop(2)
    print('1st deleted value: ' + str(deleted))
    print('deletion 1: ' + str(squares))

    # del dict[key] deletes the key-value pair and returns nothing
    del squares[4]
    print('deletion 2: ' + str(squares))

    # dict.clear() deletes every key-value pair and leaves an empty dict
    squares.clear()
    print('deletion 3: ' + str(squares))


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

Looping Over Key Value Pairs

There are a few ways to loop over the key value pairs in a dictionary.

```
def main():
    grocery_list = {'lemons': 3, 'apple juice': 1, 'spaghetti sauce': 1,
                    'noodles': 2, 'grape juice': 1, 'milk': 2,
                    'strawberries': 1, 'honey': 2}
    grocery_prices = {'lemons': 0.75, 'apple juice': 4.79,
                      'spaghetti sauce': 1.59, 'noodles': 1.99,
                      'grape juice': 5.49, 'milk': 3.99, 'strawberries': 3.99,
                      'honey': 4.99}

    # you can loop through the keys in a dictionary using dict.keys()
    total_price = 0
    for key in grocery_list.keys():
        # accessing grocery_prices[key] only works because two dictionaries
        # have identical keys
        total_price += grocery_list[key] * grocery_prices[key]
    print('grocery bill: ' + str(total_price))

    # you can also loop through the keys in a simpler way
    most_expensive_item = 'lemons'
    for key in grocery_prices:
        if grocery_prices[key] > grocery_prices[most_expensive_item]:
            most_expensive_item = key
    print('most expensive item: ' + str(most_expensive_item))

    # you can loop through the values using dict.values()
    grocery_count = 0
    for value in grocery_list.values():
        grocery_count += value
    print('number of groceries bought: ' + str(grocery_count));

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

An important thing to note about looping through keys with nested dictionaries: the keys in the inner dictionaries will not be iterated over. You have to use nested loops to get to the inner keys (nested loops for nested dictionaries).

Another important thing to note is that the functions `dict.keys()` and `dict.values()` do not return lists. What they do return is not really all that important (but it is similar to what the `range` function returns). What is important is that we have a way to turn their return values into lists when we need to: the `list` function. The `list` function is just like every other type conversion function we have learned about. You can see it in action below:

```
def main():
    grocery_list = {'lemons': 3, 'apple juice': 1, 'spaghetti sauce':
1,
                    'noodles': 2, 'grape juice': 1, 'milk': 2,
                    'strawberries': 1, 'honey': 2}

    print(list(grocery_list.keys()))
    print(list(grocery_list.values()))

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

Other Functions

```
def main():
    # dictionary of playlists (lists of song titles)
    playlists = {'throwback': ['Dynamite', 'Titanium', 'Hey Ya!', 'Rather Be'],
            'rnb': ['Sure Thing', 'This is', 'Best Part', 'Talk'],
            'international': ['Drogba (Joanna)', 'Gasolina', 'Kiminomama']
    }

    # dict.get(key) returns the value associated with key or None if the key
    # doesn't exist
    rnb = playlists.get('rnb')
    sad_music = playlists.get('sad')
    print(rnb)
    print(sad_music)

    # dict.get(key, default) returns the value associated with key or default
    # if key doesn't exist
    sad_music = playlists.get('sad', [])
    print(sad_music)

    # len(dict) returns the number of key value pairs
    print('number of pairs: ' + str(len(playlists)))


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

Just like `dict.keys()` and `dict.values()`, for nested dictionaries, the `len` function only returns the number of key value pairs in the outer dictionary.

Strings

Quest

[write a new quest]

Characters

Strings can be thought of as sequences of characters, but there isn't a *character* data type in Python. Still, it's worth understanding how individual characters are represented within a larger string.

There are so many different symbols you can write on a computer. Even though Python is written with English keywords and letters, there are plenty of other symbols out there.

Examples of Single Characters:

Symbols:

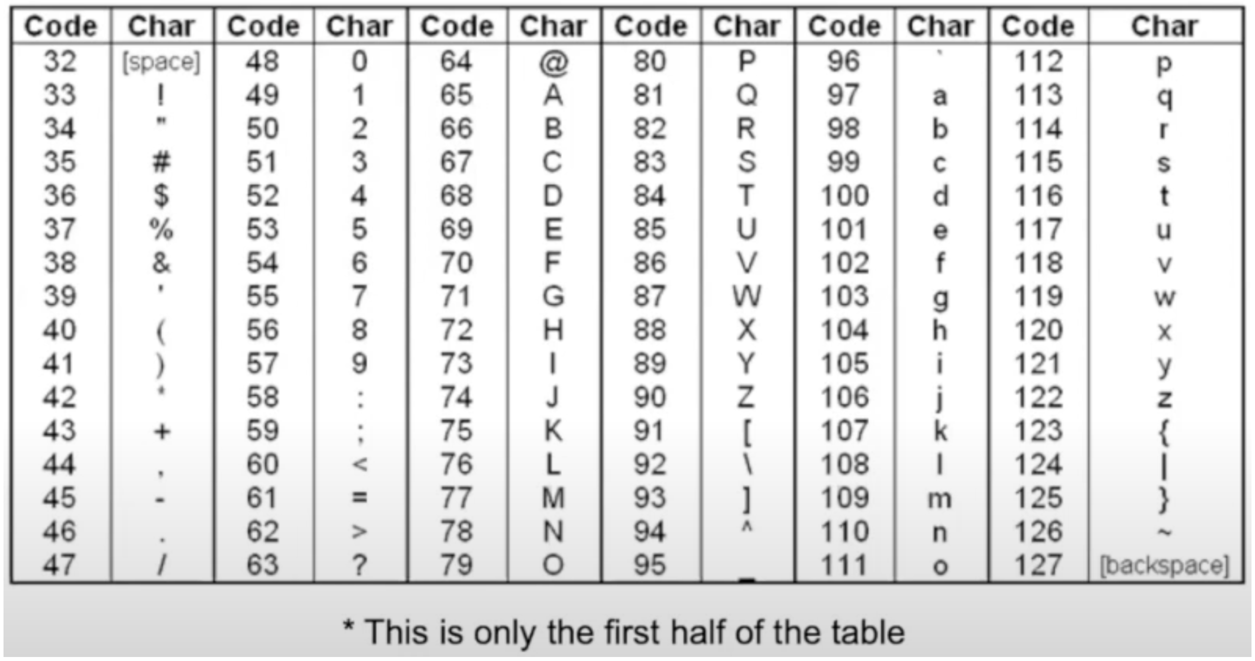
```
letter_a  = 'A'

plus      = '+'

zero      = '0'

space     = ' '

greek_pi  = 'π'

emoji     = '😎'
```

Escape Characters:

```
new_line  = '\n'

tab       = '\t'

backslash = '\\'

backslash = '\''
```

ASCII

The American Standard Code for Information Interchange, also known as ASCII, is the most popular encoding format for storing text on computers. Each character is associated with a unique number according to the following table:



Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

* This is only the first half of the table

Credit: Using portions of slides by Eric Roberts

A link to the full ASCII table can be found here.

Unicode

As you'll notice, ASCII doesn't cover nearly all of the characters you could write. There are no emojis, and letters from most other languages are missing. Even some of the characters we listed above are not included in the ASCII table. To cover a broad range of characters, Python supports Unicode, a much more robust encoding format that covers far more characters than ASCII does. The identifiers for each symbol look like *U+* followed by some letters and numbers. Here are a few examples:

Letter	Unicode
A	U+0041

+	U+002B
0	U+0030
π	U+03CO
😎	U+1F60E

Most of the time, you won't have to deal with these codes yourself. You can just copy and paste the symbol you want into your code, and Python will handle its Unicode value for you.

What is a String?

Now that we've learned about characters, we can move onto strings. Characters are the building blocks of strings. After all, way back in Intro to Python, when we first introduced you to the concept of variable types and strings, we defined them as text or character sequences between "" or ''. Let's expand on this definition. Specifically, let's expand on what we mean by *character sequence*.

Consider this string: `"Hasta la vista, baby"`

If we were to visually represent that as a sequence of characters, it would look something like this:

H	a	s	t	a		l	a		v	i	s	t	a	,		b	a	b	y

This looks kind of like a list! Strings can be thought of as a like a special type of list of characters. We can even index through strings the same way that we index through lists (using square brackets `[]`).

H	a	s	t	a		l	a		v	i	s	t	a	,		b	a	b	y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

REPL

```
$ python
>>> quote = "Hasta la vista, baby"
>>> quote[9]
v
```

We can also use colons to get string slices the same way that we get list slices. A string slice is called a substring.

```
def main():
   quote = "Hasta la vista, baby"
   first_word = quote[0:6]
   print(first_word)
if __name__ == '__main__':
   main()
```

Strings are Immutable

Now there is a reason we said that strings are *like* a kind of list. In the section on dictionaries, we mentioned that immutable types are things like ints, floats, bools and strings. We also said that lists and dictionaries are both examples of mutable types. This difference in mutability is an important one. it means that we cannot explicitly edit a string as we please. In fact, if we try to do so, we will get an error 🛑:

```
def main():
   quote = "I'm the king of the world!"
   quote[20] = 'W'
   print(quote)
if __name__ == '__main__':
   main()
```

While we can't explicitly edit strings, as we saw in variables and basic arithmetic, we can reassign or concatenate them to get the results that we want. As a reminder, string concatenation is when we 'glue' two strings together using the + operator.

```
def main():
   quote = "Where is Gamora?"
   print(quote)
   quote = "Who" + quote[5:]
   print(quote)
   quote = "Why" + quote[3:]
   print(quote)

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

Each update to `quote` is a reassignment which essentially creates a new string based on the slices and concatenation and assigns it to `quote` (as opposed to altering the original). This is why, even though strings are immutable, we can still do operations like string concatenation.

As an aside: the shortcut operation that we discussed in basic arithmetic also works for string concatenation:

```python
def main():
    quote = "Bueller..."
    quote += quote + quote
    print(quote)

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

Useful Functions Part 1

Aside from concatenation, there are also several other useful operations and functions that Python gives us for working with strings. Let's look at a couple below:

```python
def main():
    # len(str) returns the length of the given string
    quote = 'Hakuna Matata'
    print('length of', quote, '=', len(quote))

    # ord(char) takes in a single character and returns the associated unicode
    # value
    print('unicode for \'A\':', ord('A'))
    print('unicode for \'a\':', ord('a'))
    print('unicode for \'🥳\':', ord('🥳'))
    print('unicode for \'$\':', ord('$'))
    print('unicode for \'好\':', ord('好'))
if __name__ == '__main__':
    main()
```

Why do we need unicodes? Well it turns out that Python uses them for another useful operation. If you use comparison operations on strings, Python will compare the unicode values. You can use this to check string equality using `==` or to check the alphabetical order of two strings with `< >`. As we saw in the example above `'3rd '` and 'a' have different unicodes, so you must be careful when comparing strings with different cases.

```python
def main():
   # == checks the equality of two strings based on unicodes
   quote1 = "...shaken, not stirred."
   print('ex. 1', str(quote1 == '...shaken, not stirred.'))
   # case matters for equality
   print('ex. 2', str(quote1 == '...Shaken, not stirred.'))
   # < and > can be used to check alphabetical order
   exclamation1 = "excelsior"
   exclamation2 = "eureka"
   print('ex. 3', str(exclamation1 > exclamation2))
   # strings with different cases, punctuation and whitespace might interfere
   # with the accuracy of checking for alphabetical order
   quote2 = "Nobody move! I dropped me brain!"
   quote3 = "i've got a jar of dirt!"
   print('ex. 4', str(quote2 > quote3))
   # if one string is shorter than the other but equal otherwise, the shorter
   # string < longer string
   quote4 = "I don't have friends"
   quote5 = "I don't have friends, I have family."
   print('ex. 5', str(quote4 < quote5))
if __name__ == '__main__':
   main()
```

Last but not least, there is a very useful keyword in Python for determining if a string is a substring of another. We've seen this keyword before: the `in` keyword.

```python
def main():
   quote = 'Hoo-hoo! Big summer blowout!'
   print('sum' in quote) # will print true
   print('Hoo!' in quote) # will print false (case sensitive)
if __name__ == '__main__':
   main()
```

Looping over Strings

Putting a few of the things above together with our knowledge of for-each loops, we now have all of the tools we need to loop over a string!

Let's say that we want to create function to reverse a string. We could do so in three ways:

Method 1: indexing using a for loop

```python
def reverse_string(string):
   result = ""
   for i in range(len(string)):
      result = string[i] + result
   return result
def main():
   quote = 'You know what kind of plan never fails? No plan at all'
   print(reverse_string(quote))
if __name__ == '__main__':
   main()
```

Method 2: for each loop

```python
def reverse_string_v2(string):
   result = ""
   for ch in string:
      result = ch + result
   return result
def main():
   quote = 'I feel the need... the need for speed!'
   print(reverse_string_v2(quote))
if __name__ == '__main__':
   main()
```

As you can see, you can loop over strings the same way that you lop over lists. Both for and for each loops work well and which one you choose depends on whether or not you want to know the index of a particular character as well.

We did mention that there are 3 methods to reversing a string. Is there a third loop? 🤯 No, but there is a clever way to use slice indexing to achieve the same result:

Method 3: (not a loop) fancy indexing

```python
def reverse_string_v3(string):
   '''
   This uses the slice operator in a special way. With no
   start, no end and a delta of -1, slice reverses.
   '''
   return string[::-1]
def main():
   quote = 'success के पीछे मत भागो, excellence के पीछे भागो'
```

```python
    english_translation = "Don't run after success, run after excellence"
    print(reverse_string_v3(quote))
    print(reverse_string_v3(english_translation))
if __name__ == '__main__':
    main()
```

Useful Functions Part 2

These are several more functions that we feel are useful to know for working with strings. Unlike the previous functions and operators, all of these are *string functions* not just functions that use strings as arguments. These are divided into two parts: **must know** and **good to know**. The goal here is not memorization. Feel free to return to this section whenever you need some insight into which functions are available to you.

Must know

```python
def main():
    # str.split(separator) returns a list of substrings
    # substrings are determined by the separator
    quote = 'We have so much to say, and we shall never say it.'
    print('split:', quote.split(' '))
    # str.upper() returns str in all uppercase
    quote = 'Do or do not. There is no try.'
    print('upper:', quote.upper())
    # str.lower() returns str in all lowercase
    print('lower:', quote.lower())
    # str.replace(oldsubstr, newsubstr) replaces all instances of oldsubstr
    # with newsubstr in str
    quote = "What's done is done when I say it's done."
    print('replace:', quote.replace('done', 'good'))
    # str.find(substr) returns the index of the first instance of a substr
    quote = "You didn't ask for reality; you asked for more teeth!"
    print('find:', quote.find('for'))
    # str.strip() leading and trailing whitespace
    quote = '   Sometimes your whole life boils down to one insane move    '
    print('strip:', quote.strip())


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

Good to know

```
def main():
   # str.startswith(substr) returns true if str begins with substr
   quote = "Thrones are for Decepticons. Besides, I'd rather roll."
   print('startswith:', quote.startswith('Th'))
   # str.endswith(substr) returns true if str ends with substr
   quote = 'We could all have been killed... or worse, expelled.'
   print('endswith:', quote.endswith('end'))
   # str.title() returns str with the first letter of each word capitalized
   quote = "Give it up, Sid. You know humans can't talk"
   print('title:', quote.title())
   # str.isalpha() returns true if every character is alphabetic
   print('isalpha1:', 'Hello'.isalpha())
   print('isalpha2:', 'Code In Place'.isalpha())
   # str.isdigit() returns true if every character is a numerical digit
   print('isdigit:', '173'.isdigit())
   # str.isspace() returns true if every character is whitespace
   print('isspace:', '  '.isspace())

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

Worked Example - is_palindrome

We just threw a lot of new functions at you, so we thought it might be helpful to see a few of those functions in action. Let's say you wanted to implement a function that could check to see if a given string is a palindrome.

For context, a palindrome is any string that is the same forwards and backwards such as `'abba'`, `'racecar'`, or `'kayak'`.

We can use our reverse function to implement it:

```
def reverse_string(string):
   result = ""
   for ch in string:
      result = ch + result
   return result
def is_palindrome(string):
   rev = reverse_string(string)
   return string == rev
def main():
   # This is a palindrome! It should return true.
   print(is_palindrome('Mr. Owl ate my metal worm.'))
   # We changed Mr to My. This should return false.
   print(is_palindrome('My Owl ate my metal worm.'))
if __name__ == '__main__':
   main()
```

Wait a minute! Why does the first call to `is_palindrome` not return `True`? Well, as we've said before, things like whitespace, punctuation, and case all matter when using `==` to check for equality. We need to find a way to get rid of all the extra characters and standardize the case. Let's use some string functions!

```
def reverse_string(string):
   result = ""
   for ch in string:
      result = ch + result
   return result
def normalize(string):
   normalized = ''
   for ch in string:
      if ch.isalpha():
         normalized += ch
   return normalized.lower()
def is_palindrome(string):
   normalized = normalize(string)
   rev = reverse_string(normalized)
   return normalized == rev
def main():
   # This is a palindrome! It should return true.
   print(is_palindrome('Mr. Owl ate my metal worm.'))
   # We changed Mr to My. This should return false.
   print(is_palindrome('My Owl ate my metal worm.'))
if __name__ == '__main__':
   main()
```

Yay! We now have a working program.

One last thing before you move on. Most of the strings in this section (and all of the ones assigned to quote) are quotes from famous movies. Try to see how many movies you can figure out!

Tuples

Quest

We have reached our final container of the Containers section: Tuples! If your first thought when you see that word is "what in the world is a tuple?" don't worry. Tuples are pretty simple containers. You can think of a tuple as a list that you can't change. This means that whatever we assign to the tuple at it's declaration is the tuple's final value.

We declare a tuple using parentheses `()`:

```
names = ('Sabeen', 'Justin', 'Mukesh', 'Chen', 'Lucia', 'Devon', 'Yousaf',
'Aisha')
```

and access values using square brackets `[]` and indexes just like with lists:

```
names[0]
```

If you remember our brief discussion on mutability from the last section, tuples are another example of an immutable type. If you try to change the value of an element tuple, you will get an error 🛑:

```
def main():
    names = ('Sabeen', 'Justin', 'Mukesh', 'Chen',
          'Lucia', 'Devon', 'Yousaf', 'Aisha')

    names[4] = 'Edwin'      # this line will error
    print(names[4])

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

You might be wondering why we would want to use tuples. Why use a list that has less functionality? It turns out there are several things that make tuples a useful container in Python.

1. Safety: If we know the values that we want to store in a list ahead of time, it might be safer to store them in a tuple. That way, we know that the values will never be altered accidentally (like by a helper function).

2. Tuples can be dictionary keys: In the last section, we mentioned that dictionary keys must be immutable. This means that lists and dictionaries can't be keys, but what if we want to make a group of values the key? We can just use tuples.

3. Returning multiple things: Tuples also have another special use. Remember back in Multiple Returns when we said that there was a way to return multiple things at once? We meant tuples! Check out this example below:

This is a program to find the equation of a linear line based on coordinates from two points. Don't worry if you are not familiar with the algebra that goes into this. All you need to know is that the equation of a line requires two things: the slope and the intercept. We give you the equations for both in the program below.

```
def get_equation(x1, y1, x2, y2):
   slope = (y2 - y1) / (x2 - x1)
   intercept = y1 - (slope * x1)

   return slope, intercept

def main():
   slope, intercept = get_equation(1, 1, 2, 4)
   print('slope:', slope)
   print('intercept:', intercept)

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

As you can see above, all that is required to return multiple things is a comma separating each term. Both things are outputted by the function. It is important to note that when setting variables equal to the result of the function call, the number of variables on the left side of the equal sign should match the number of things returned by the function. If you have too many variables, an error will occur 🛑.

```
def get_equation(x1, y1, x2, y2):
   slope = (y2 - y1) / (x2 - x1)
   intercept = y1 - (slope * x1)

   return slope, intercept

def main():
   slope, intercept, distance = get_equation(1, 1, 2, 4)
   print('slope:', slope)
```

```
    print('intercept:', intercept)
    print('distance:', distance)

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

Ok, at this point you may be wondering what exactly all of this has to do with tuples. Well let's see what happens if we only put one variable on the left side of the assignment operator.

```
def get_equation(x1, y1, x2, y2):
    slope = (y2 - y1) / (x2 - x1)
    intercept = y1 - (slope * x1)

    return slope, intercept

def main():
    equation_vars = get_equation(1, 1, 2, 4)
    print('slope and intercept:', equation_vars)

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

As you can see, when you return multiple things you are really just returning a tuple! It is almost like there are invisible parentheses around `slope, intercept` in the return statement of `get_equation`.

So, tuples provide safety for storing values we don't want to change and allow us to return multiple values from functions. Now that you have a sense for what they are and what they do, you have officially learned all three of the containers for Code in Place! You rock!

File Reading

Now that we have all of this knowledge on containers, let's apply it! There are many times when as a programmer, you will want access to outside data. Outside data can come from from the user (as is the case with the input function) or it can come from files. Files can be anything from plain-text (characters and words on a page) to music (like an mp3 file), but for this class we are going to focus on plain-text.

There are two ways to read a file in Python. Let's look at Method 1.

```
with open('mydata.txt') as file:
    for line in file:
        line = line.strip() # optional
```

The `with` keyword creates an association between the name of the file `'mydata.txt'` and the variable file. The `as` keyword helps indicate which variable will be associated. The actual opening of the file is handled by the `open` function which takes in the name/path of the file to be opened.

Once the file is opened, the for each loop grabs each line in the file as a string with a new line character `'\n'` on the end. Usually, when processing file data, we don't want to deal with a new line character, so we can use the strip method to get rid of it.

At the end of the `with` block (when the indentation stops), the file is automatically closed.

Now for Method 2.

```
for line in open('mydata.txt'):
    line = line.strip() # optional
```

Method 2 is very similar to Method 1. We merely skip the association step of the file with a variable. This is a newer method that is becoming more and more popular amongst programmers. One advantage of this is that the code tends to finish faster. Most programs that you've written for Code in Place have run almost instantly, but with large files and complex programs, run time becomes more of an issue. The tradeoff is that the file is not automatically closed at the end of the for loop. It will remain open for the rest of the program and then close once the program is complete. This is not an issue if you do not plan to use the file later in the program, so it's up to you which method you would like to use!

Worked Example - get_word_counts

Now that we've seen both methods, let's put one to use! Let's say we want to build a program that counts the number of times each word is used in a particular file and stores it in a dictionary. For example: let's say we had a file like this:

`sample.txt`

> *This is a file with some words. This file can be used to*
>
> *count words. THIS IS SO COOL!!*

We would want to output to be this:

```
{ 'this' : 3, 'is' : 2, 'a' : 1, 'file': 2, 'with': 1, 'some': 1, 'words': 2,
'can': 1,
 'be' : 1, 'used': 1, 'to': 1, 'count' : 1, 'so': 1, 'cool': 1}
```

Let's see how this would be implemented below:

```
PUNCTUATION = '.!?,-:;'

def delete_punctuation(string):
    '''
    Removes punctuation characters from a string and returns the resulting
string
    (without punctuation).
    '''
    result = ''
    for char in string:
      # check char is not a punctuation mark
      if char not in PUNCTUATION:
          result += char
    return result


def get_word_counts(file_name):
      '''
      reads file and returns a dictionary with the words in the file and the
number of
        occurances of each word
```

```
       '''
       counts = {}

       with open(filename) as file:   # open file to read
           for line in file:
                 words = delete_punctuation(line).split()
                 for word in words:
                  word = word.lower()  # make all words lower case
                      if word not in counts:
                    counts[word] = 1
                  else:
                    counts[word] += 1
       return counts


def main():
    print(get_word_counts('sample.txt')


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

If you haven't already noticed, a lot of file reading is just text or string processing! That is why the first function that we write is `delete_punctuation`:

```
PUNCTUATION = '.!?,-:;'

def delete_punctuation(string):
    '''
    Removes punctuation characters from a string and returns the resulting
string
    (without punctuation).
    '''
    result = ''
    for char in string:
      # check char is not a punctuation mark
      if char not in PUNCTUATION:
          result += char
    return result
```

We use the global constant `PUNCTUATION` and check every character in the string to build up `result`. Only characters not in `PUNCTUATION` are added to result effectively removing the punctuation characters.

Then we get to our key function: `get_word_counts`. The first three lines should look pretty familiar. We declare an empty dict, open the file and start looping over the lines.

```
def get_word_counts(file_name):
    ...
    counts = {}

    with open(filename) as file:    # open file to read
        for line in file:
```

Next, we use our helper function to get rid of the punctuation and obtain a list of the resulting words using the `split` function. One thing to note is that if you don't give the `split` function an argument, it defaults to using whitespace as the separator.

```
words = delete_punctuation(line).split()
```

We loop over each word in the list of words and convert it to lowercase to make sure that our dictionary's keys are not case sensitive (`'THIS'` should not have a separate count from `'This'`).

```
for word in words:
    word = word.lower()
```

Last but not least, we add the key to our counts dictionary if it is not already in there and increment the count by one if it is. Then, we return the completed dictionary.

```
          if word not in counts:
              counts[word] = 1
          else:
              counts[word] += 1
           return counts
```

When you break it down into simple steps, file reading becomes really simple! All you need is the right text processing tools. Remember that you can always refer back to the strings section if you are unsure of what functions you have available.

— Memory —

Intro to References

Quest

A magician never reveals their tricks, right? Well, we're not magicians, we're computer scientists. While a lot of what we do might, at times, feel like magic, there is always a logical explanation for why our programs do what they do. So, with that in mind, we are going to show you one of our tricks. It's time to peek behind the curtain and see what's really going on with variables and mutability!

The URL Analogy

Right now, you are reading a document we wrote to teach you about Python. Where does this document live? You might be reading this offline or in some other format, but many readers found this page using a link on the Code In Place website. The course reader lives online; it has a special URL or an address you can type into the internet that leads you to this page.

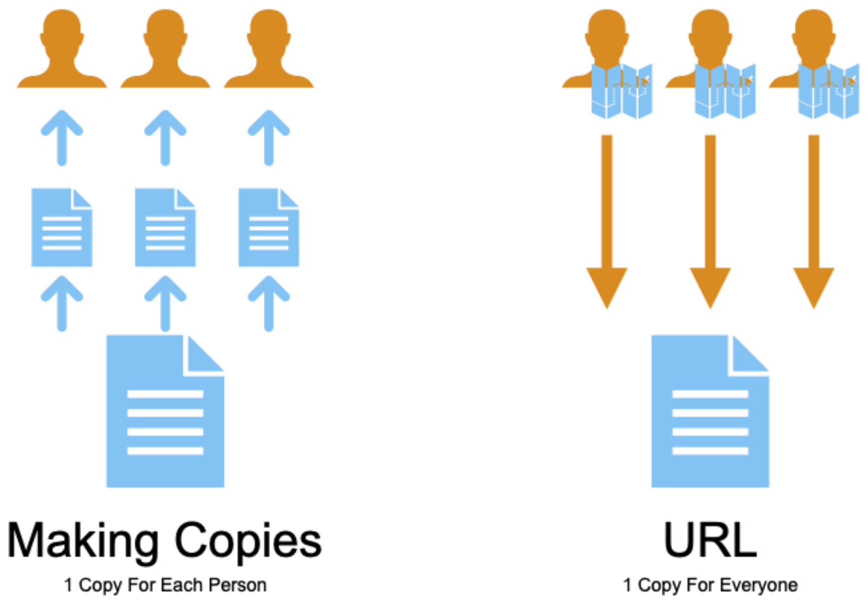
Why do you think we chose to upload this course reader to the internet? We could've just sent a copy to every student in Code In Place, but there are two problems with that idea:

1. There are thousands of students taking Code In Place (*side note: How amazing is that!)* We'd have to make a copy for every single student and send it to them individually.
2. Say we wanted to fix a typo or add some additional content to the page. To make sure everybody has the most up-to-date version, we'd have to make thousands of new copies, again and again, each time we want to make a change.

What a nightmare! Think about how much more efficient using a URL is!

1. Instead of making thousands of identical copies of the same information, there is just one document stored on our website.
2. Instead of sending you all the pages of information in this course reader, we just send you a short URL.

When we edit the course reader, nobody needs a new copy sent to them. Since they are coming to *our* website and looking at *our* document, the changes have already been made by the time they come knocking on our door.



Why are we telling you all this? The key idea is that it is much more efficient to tell someone where information is stored than to make a copy of that information and send it to them. Programmers realized this and decided it was worth building Python in a way that supports this idea of a URL. In Python, these URLs are called references.

Memory

You have a brain. Part of that brain is responsible for helping you remember things. In a lot of ways, your computer is like a brain too. The part of your computer that remembers things is called memory. If you recall all the way back to the variables section, we told you that your computer has these little suitcases called memory addresses that store bits of information. When you assign a new variable, your computer picks an empty one of these suitcases and stores whichever value you gave inside that suitcase. That variable now serves as a luggage tag to help the computer identify where it stored your information so that it can retrieve it for you later.

References

It is time for us to reveal the magic trick! Drum roll please…

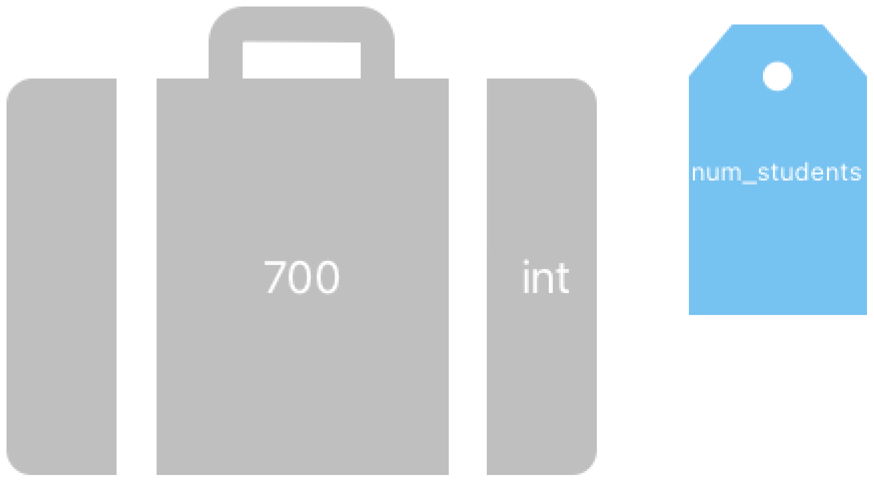
Variables don't actually store values. Rather, they just keep track of which memory address the value is stored at. Variables really just store a reference to the value because they *refer* your computer to the correct suitcase where the value is actually stored.

We do this for the precise reasons stated up above, and references become increasingly useful the more data you have. If you had a list of, say, 10,000 elements, passing a short memory address as an argument takes up far less space than copying the entire list and passing that instead. No matter how big your data gets, references will always stay the same size, making it easy to pass massive amounts of data between functions without much of a hassle! Also, if multiple variables reference the same list, editing the list for one variable edits it for all of them.

Let's go back to a variable example to understand how they *really* work:

```
num_students = 700
```

We told you that you could think of `num_students` like a piece of luggage!



The image above was first shown in the variables section, though perhaps it makes more sense now, given what we just learned. The "luggage" tag -- the variable *num_students* -- doesn't actually store the value 700. Instead, *num_students* tells you where to find the correct suitcase.

How Variables Act

In reality, all variables in Python store a reference to the data they are supposed to keep track of. When you pass a variable in as an argument, the information received by the corresponding parameter is actually just the reference to whatever value that argument stores.

Sometimes, though, it really does feel like the value itself is being passed into the function. As it turns out, some of the data types you are more familiar with do act as if that is true, even if references are really what is going on under the hood.

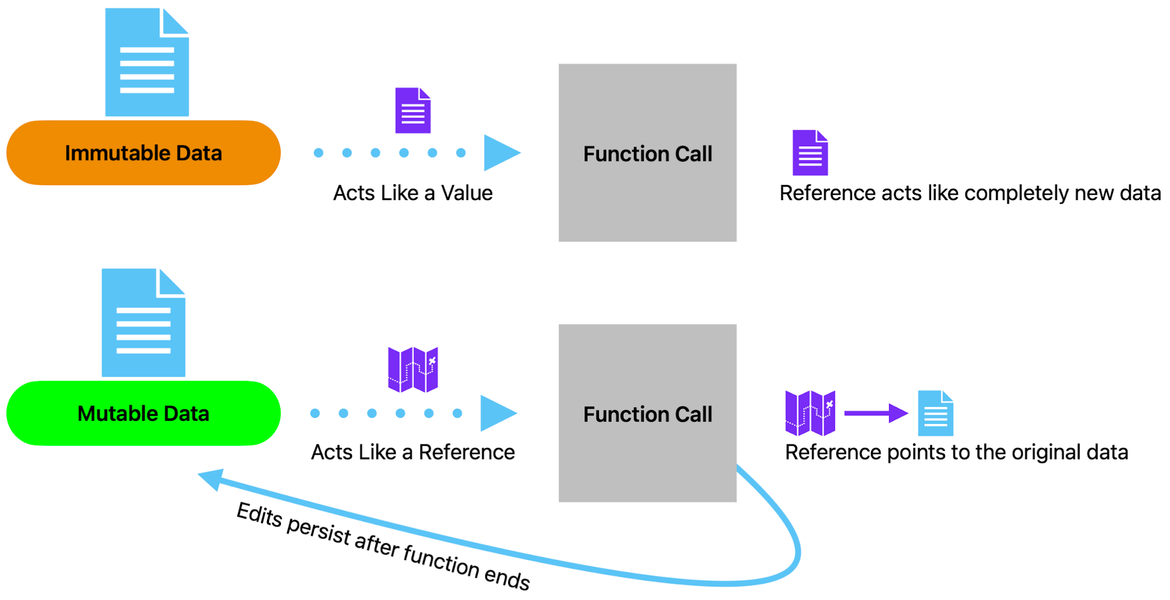
All data types in Python are passed by reference, meaning the memory address is what actually gets passed. Here’s why this is important: if you pass a reference to a value as an argument and then edit that value within the function, that value has now been permanently changed. Using our example from before, this would be like editing our one shared copy of the textbook that everybody is reading. This behavior shows up most obviously with mutable data types. If I pass in a reference to a list and then change that list somehow, the original list gets changed too:

```python
def add_ending_to_list(list):
   '''
   this function takes in a reference to a list and appends a
   final element to the end
   params:
      list : reference to a list
   '''
   list.append("End of List")
def main():
   # We create a list of integers and store a reference to it
   # in the variable called numbers
   numbers = [1,2,3,4,5]
   # Then, we pass that reference to our helper function
   add_ending_to_list(numbers)
   # After the helper function is done, numbers still references
   # the same location in memory, but the list stored there has
   # been modified
   print(numbers)
if __name__ == "__main__":
   main()
```

Notice how add_ending_to_list doesn’t return anything. Furthermore, notice how we just called add_ending_to_list by itself. Unlike the x = change(x) paradigm, dealing with mutable data just involves passing that data in and then modifying it how you wish. Once the function ends, your variable will already reference the newly updated information. The reason pass by reference works this way is that the argument is still referencing the same location in memory, but the value at the location in memory has changed.

What about immutable data types? While they are also passed by reference, immutable data types *seem* like they are **passed by value**.

If you pass an integer in as an argument and then modify the value of the parameter, the value of the variable that was passed in is unaffected. Why this happens exactly is less important, but you just need to remember that you can't modify the value of an immutable argument within a function and expect that argument to have changed:


Immutable Data
Acts Like a Value
Function Call
Reference acts like completely new data
Mutable Data
Acts Like a Reference
Function Call
Reference points to the original data
Edits persist after function ends

```python
def double_it(number):
   '''
   this function takes in a reference to a number and doubles
   its value
   '''
   number *= 2
def main():
   # We store a 4 in memory and assign a reference to that 4 to
   # the variable called number
   number = 4
   # Then, we pass that reference to our helper function
   double_it(number)
   # Even though we passed number by reference, its value is
   # not effected by what happens inside the function
   print(number)
```

```python
if __name__ == "__main__":
    main()
```

Important: We say immutable types "seem" to be passed by value because they are actually passed by reference. Don't forget! Remember, you can refer to the table of mutable and immutable types in the section on mutation*.*

Objects

Quest

Now that we have a basic understanding of references, we are going to take a brief detour back to the graphics section. Remember when we had you create a canvas to display all of the cool graphics that you made? We had you do so like this...

```
canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)
```

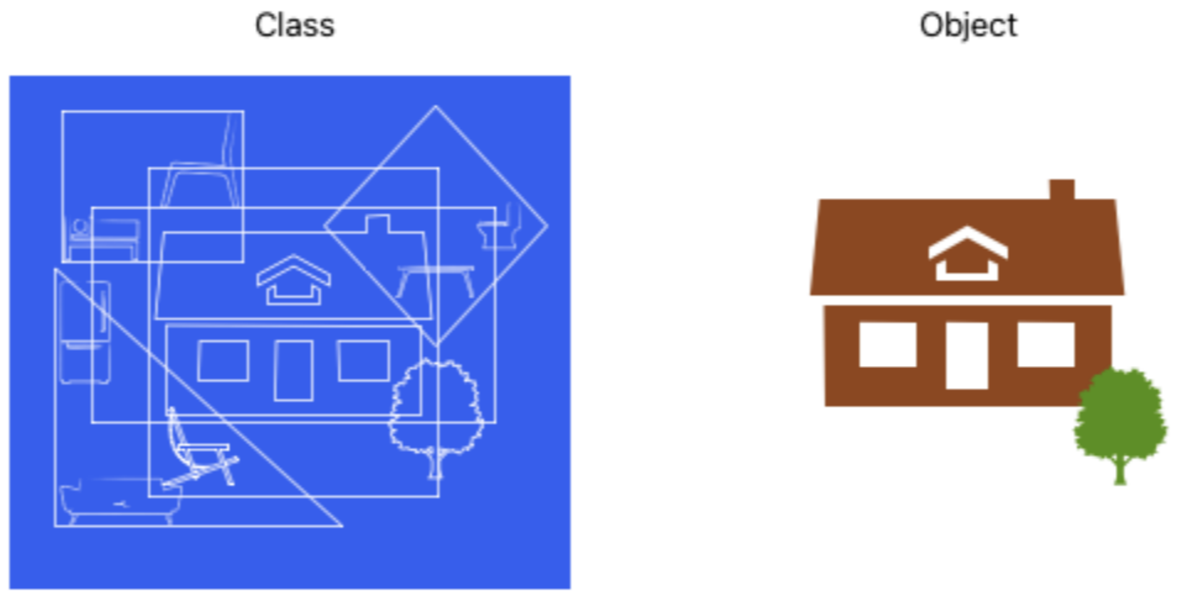
...and then when you wanted to add something to the canvas, you would do something like this:

```
canvas.create_line(10, 20, 100, 50, 'red')
```

We ignored the weirdness of this code earlier because you didn't need to know what was going on to create the graphics. But, what is going on? What variable type is `canvas`? It is certainly not any of the types we have talked about before (int, float, bool, string, list, dictionary, tuple). Somehow, we can do functions on it using a period?! And why does the `Canvas` function start with a capital letter?

Well it turns out that canvas is a special data type called an object. Before you can understand objects, you first need to know what classes are. Classes are essentially groups of data (called instance variables) and functions (called methods). You have already seen classes before. Lists and dictionaries are built in Python classes. List functions like `index` or `remove` are really list *methods*. The cool thing about classes is that we can create our own. For example, `Canvas` is a class created by the wonderful coders at Code in Place to be able to display graphics.

Objects are *instances* of classes. Every time you create a list or dictionary variable, you are creating an instance of the list or dictionary class. You can essentially think of a class as a blueprint of a specific house and an object as a physical house that is built with the blueprint. There can be many houses built with the same blueprint and we can have many different instances or objects of one class.



In the case of graphics, `canvas` is an object of the `Canvas` class. `create_line` is a method in the `Canvas` class that we can call on the `canvas` object. To call an object's method, we use the following notation that you have seen before:

```
object_name.method_name(arg1, arg2...)
```

The period in between the object and its method is called the dot operator. It ties the method from the particular class on the right to the specific object on the left. The same notation is used for instance variables:

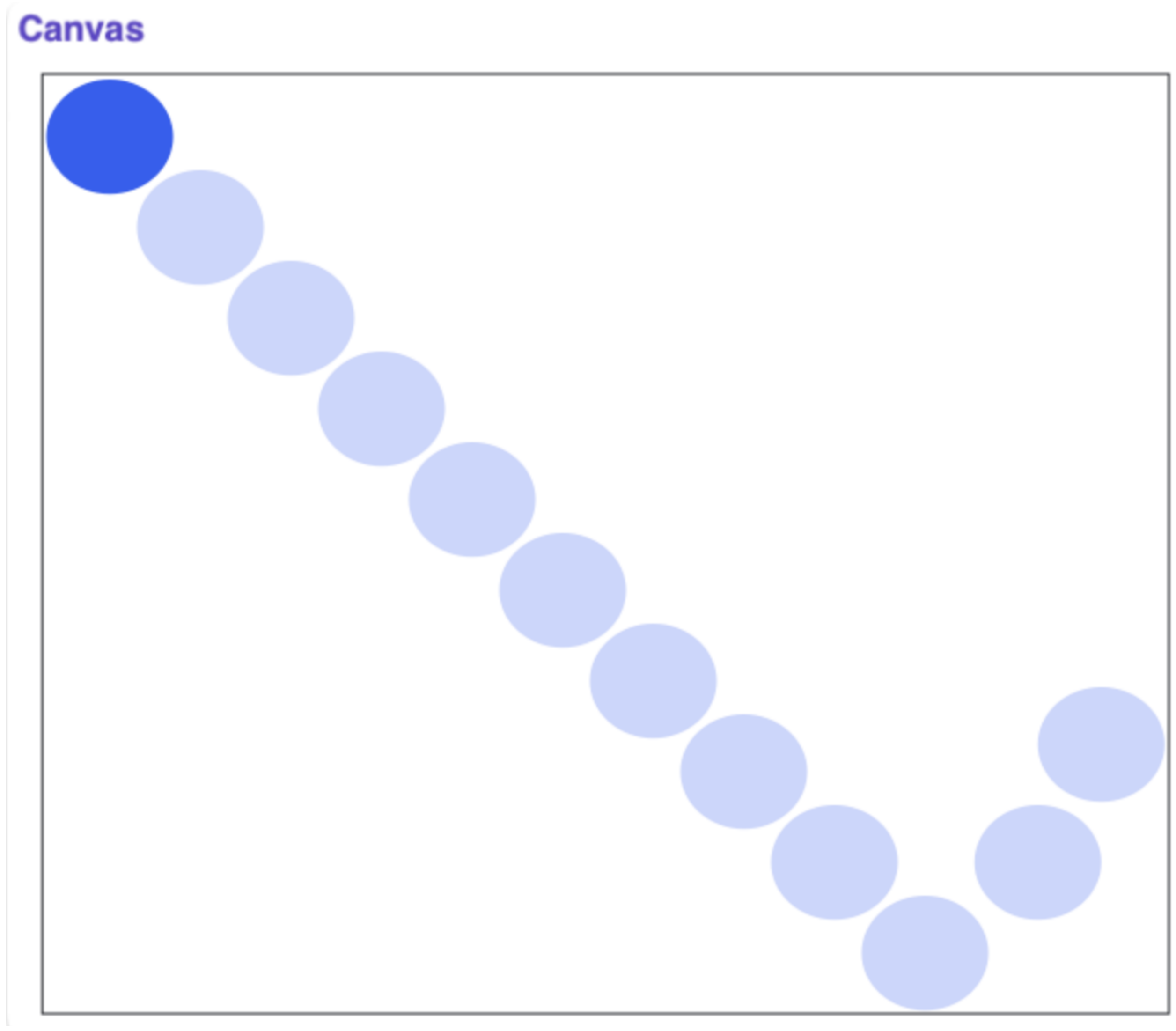
```
object_name.instance_variable
```

Note: even though string functions like strip also use the dot operator, strings are not objects. Strings are immutable whereas objects are mutable.

Mutation

As we just noted above, objects are all mutable. This means that when you pass them in as parameters to a function, the changes that you make in the helper function will affect the object in the caller function.

Let's go back to our bouncing ball program to see this in action. If you remember from the animation section, the goal was to create a program where a bouncing ball would start at the top of the screen and "bounce" every time it hit an edge.



The code adapts our bouncing ball code from before to add in a helper function.

```
from graphics import Canvas
import time
BALL_SIZE = 50
CANVAS_WIDTH = 550
CANVAS_HEIGHT = 450
DELAY = 0.001       # seconds to wait between each update
START_X = 0
START_Y = 0
def make_ball(canvas): # make canvas object a parameter
   '''
   adds bouncing ball to the given canvas and returns it
   '''
   return canvas.create_oval(START_X, START_Y, BALL_SIZE, BALL_SIZE, 'blue')
```

```python
def main():
    # setup
    canvas = Canvas(CANVAS_WIDTH, CANVAS_HEIGHT)
    ball = make_ball(canvas)
    change_x = 1
    change_y = 1
    cur_x = START_X
    cur_y = START_Y

    # animation loop
    while(True):
        # change direction if ball reaches an edge
        if cur_x < 0 or cur_x + BALL_SIZE >= CANVAS_WIDTH:
            change_x = -change_x
        if cur_y < 0 or cur_y + BALL_SIZE >= CANVAS_HEIGHT:
            change_y = -change_y

        # update the ball
        canvas.move(ball, change_x, change_y)
        cur_x += change_x
        cur_y += change_y

        # pause
        time.sleep(DELAY)


if __name__ == "__main__":
    main()
```

As you can see, when canvas is passed in as a parameter, calling `create_oval` still displays an oval on the original canvas.

Mutation

Quest

Welcome to your last section in the Python reader! It has been a wild ride through the basics of Python and computer programming. We hope that you leave this reader with a strong foundation in coding and a desire to keep learning more! With that said, it's that time. The moment you have all been waiting for when we finally fully explain mutability! Before we dive into the full explanation, let's review what we already know:

There are two categories of variable types: immutable and mutable.

Immutable	Mutable
Ints	Lists
Floats	Dictionaries
Bools	Objects (all object types)
Strings	
Tuples	

Immutable types cannot be edited or mutated but are reassigned values. Mutated types can be edited. This affects how both of these variable types can be referenced. Immutable types are not changed by references such as assigning another variable to them or passing them in as arguments to functions. On the other hand, if a mutated type is assigned to another variable or passed into a function and then updated, the original variable will be affected as well.

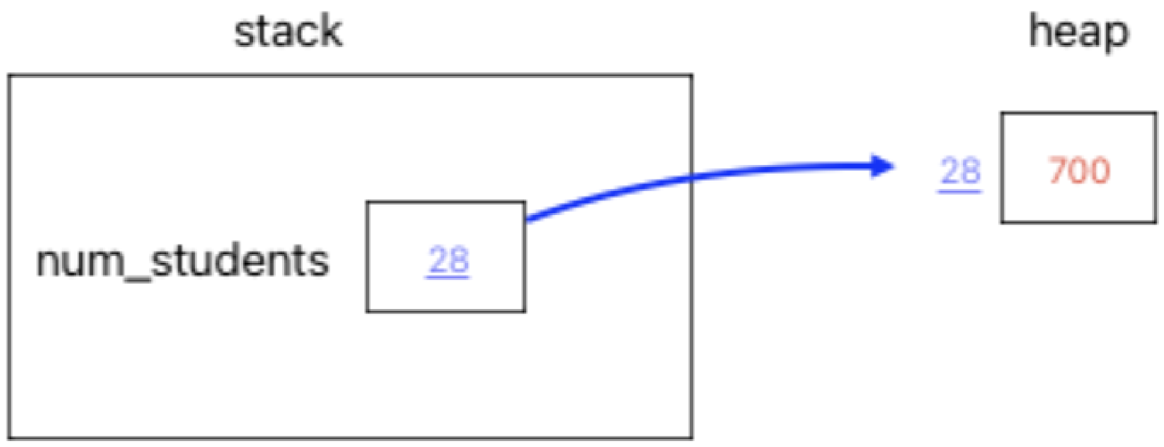
Stack vs Heap

The memory stack and heap are not super important for you to fully understand right now. We are going to give you a very low level overview to help make mutability and the url analogy easier to understand. There are two general structures of memory that programs use. The first is called the stack and it holds memory for the local variables (addresses) that you create in functions for a single program. The second is called the heap and it holds memory (values) for all programs running on your computer.

Let's visit our example from way back in the variables section again.

```
num_students = 700
```

As we said in the references section, the variable `num_students` is actually keeping track of the memory address of the value `700`. `num_students` (with the stored memory address) is stored on the stack. The memory address is actually the address or location of the value on the heap. In the example below, `28` represents the memory address.

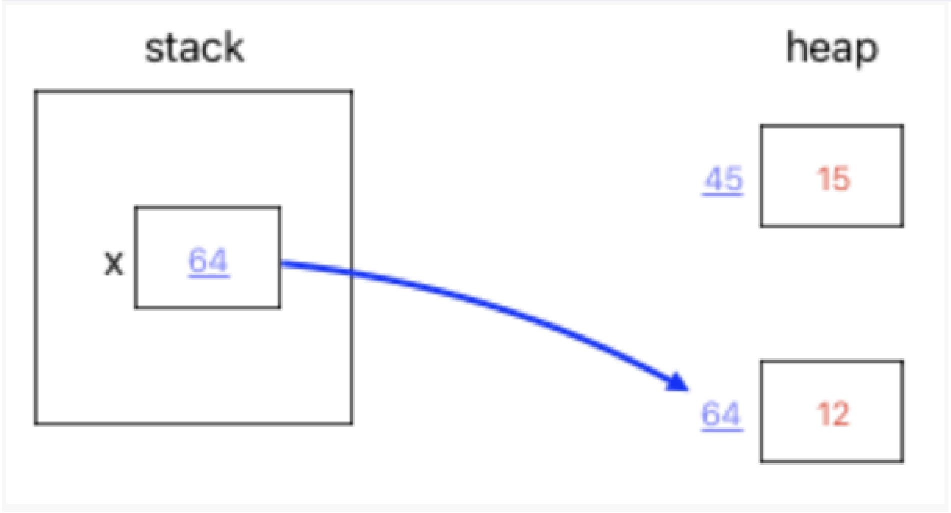


You might be wondering why we have two different areas of memory. Aside from allowing for mutability and immutability, it is also just a more efficient way to use memory. We can delete local variables from the stack once a function is over in order to save space, but we also want a separate space to make sure that things like global constants are not erased after the end of the first function.

Reassignment vs Mutation

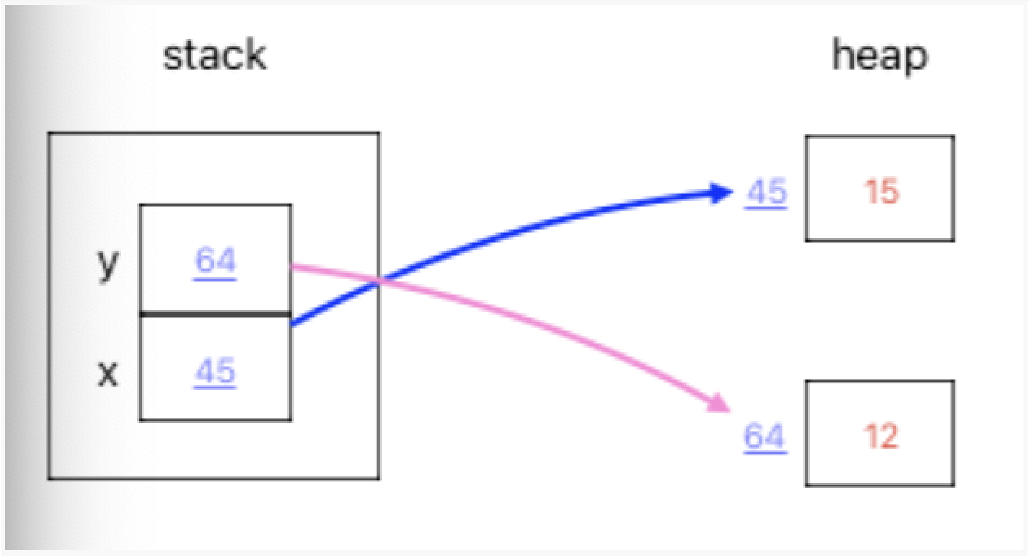
Using our new depictions of the stack and the heap, we can now tell you what goes on behind the scenes for immutable and mutable variables. When you reassign a value to a variable, what you are really doing is changing the memory address that is stored in that variable:

```
x = 15
x -= 3
```



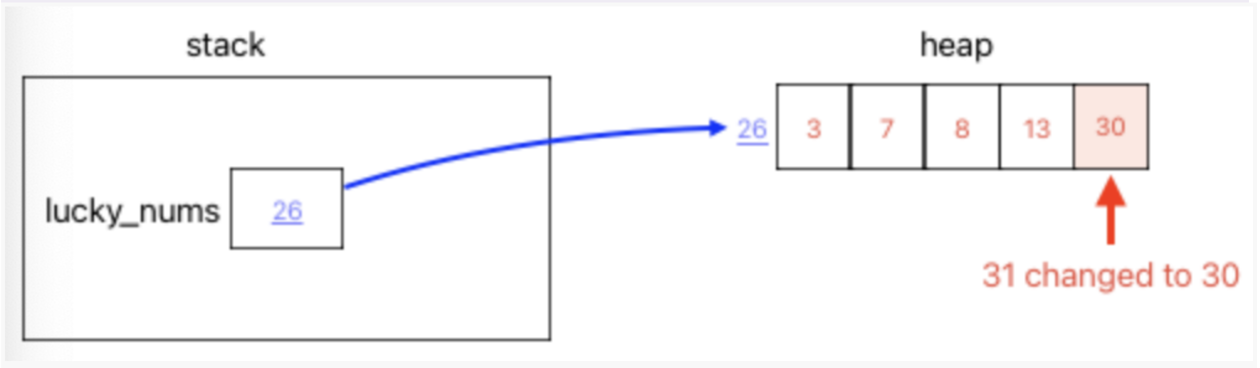
This why we can do things like set a second variable `y` equal to `x` or pass `x` in as an argument to a function and any changes to the parameter in the function or to `y` will not affect `x`. `y` or the parameter will simply be assigned a new memory address with the new value, and the value at the memory address that `x` points to will remain unchanged.

```
x = 15
y = x
y -= 3
```



That is the case for immutable variables. What happens in memory for mutable types? As you might've guessed, with mutable variables like lists, edits to the internal values actually alter the memory on the heap.

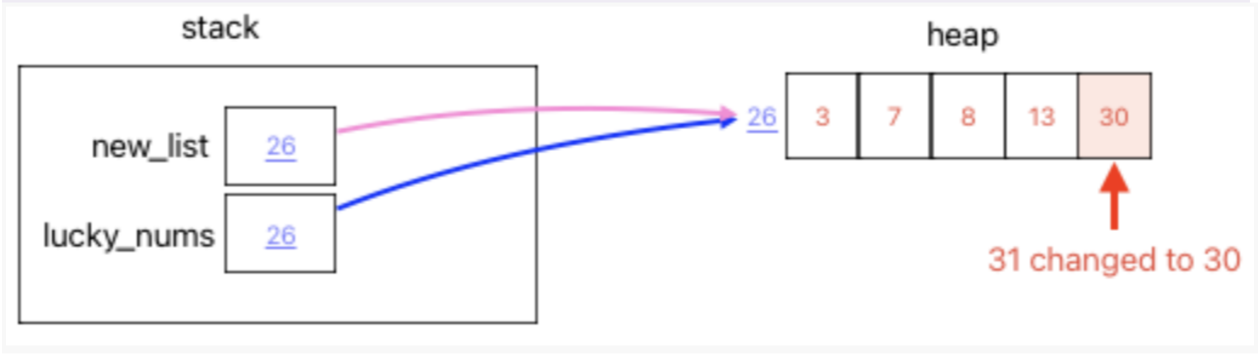
```
lucky_nums = [3, 7, 8, 13, 31]
lucky_nums[4] -= 1
```



This means that when we set a new list equal to `lucky_nums` or pass `lucky_nums` in as an argument and then make changes, the changes affect `lucky_nums` as well.

```
lucky_nums = [3, 7, 8, 13, 31]
new_list = lucky_nums
```

```
new_list[4] -= 1
```



That's it for the Python reader. You have completed your final quest with us and reached the top of the tall, beautiful mountain, but don't be sad! The end of one journey is the beginning of another. There is so much more to learn in Python and Computer Science as a whole! We have enjoyed being your tour guide up until now, but where you go next is up to you!