More About Python

Saturday, May 1, 2021 3:36 AM

More About Python

More About Python

Using Python on your own

The best way to learn any programming language is to practice it on your own as much as you can. If you have Python installed on your computer, you can execute the interpreter by running the python3 command (or just python on Windows), and you can close it by typing exit() or Ctrl-D.

If you don't already have Python installed on your machine, that's alright. We'll explain how to install it in an upcoming course.

In the meantime, you can still practice by using one of the many online Python interpreters or codepads available online. There's not much difference between an interpreter and a codepad. An interpreter is more interactive than a codepad, but they both let you execute code and see the results.

Below, you'll find links to some of the most popular online interpreters and codepads. Give them a go to find your favorite.

- https://www.python.org/shell/
- https://www.onlinegdb.com/online_python_interpreter
- https://repl.it/languages/python3
- https://www.tutorialspoint.com/execute_python3_online.php
- https://rextester.com/l/python3_online_compiler
- https://trinket.io/python3

Additional Python resources

While this course will give you information about how Python works and how to write scripts in Python, you'll likely want to find out more about specific parts of the language. Here are some great ways to help you find additional info:

- Read the official Python documentation.
- Search for answers or ask a question on Stack Overflow.
- Subscribe to the Python tutor mailing list, where you can ask questions and collaborate with other Python learners.
- Subscribe to the Python-announce mailing list to read about the latest updates in the

language.

The official language reference.

From <https://www.coursera.org/learn/python-crash-course/supplement/QWK3z/welcome-to-the-course>

Python history and current status

Python was released almost 30 years ago and has a rich history. You can read more about it on the History of Python Wikipedia page or in the section on the history of the software from the official Python documentation.

Python has recently been called the fastest growing programming language. If you're interested in why this is and how it's measured, you can find out more in these articles:

- The Incredible Growth of Python (Stack Overflow)
- Why is Python Growing So Quickly - Future Trends (Netguru)
- By the numbers: Python community trends in 2017/2018 (Opensource.com)
- Developer Survey Results 2018 (Stack Overflow)

From <https://www.coursera.org/learn/python-crash-course/supplement/qQ4ox/more-about-python>

Course code pad

Saturday, May 1, 2021 4:29 AM

https://www.coursera.org/learn/python-crash-course/quiz/q9yr3/practice-quiz-hello-world/attempt

ASCII

Friday, June 30, 2023 3:01 PM

Uppercase		Uppercase		Lowercase		Lowercase	
Unicode #	Character	Unicode #	Character	Unicode #	Character	Unicode #	Character
65	A	78	N	97	a	110	n
66	B	79	O	98	b	111	o
67	C	80	P	99	c	112	p
68	D	81	Q	100	d	113	q
69	E	82	R	101	e	114	r
70	F	83	S	102	f	115	s
71	G	84	T	103	g	116	t
72	H	85	U	104	h	117	u
73	I	86	V	105	i	118	v
74	J	87	W	106	j	119	w
75	K	88	X	107	k	120	x
76	L	89	Y	108	l	121	y
77	M	90	Z	109	m	122	z

1. Introduction to programming

Why do we need to learn the syntax and semantics of a programming language?
To allow us to clearly express what we want the computer to do

What's automation?
The process of replacing a manual step with one that happens automatically

Which of the following tasks do you think are good candidates for automation? Check all that apply.

- Installing software on laptops given to new employees when they are hired
- Periodically scanning the disk usage of a group of fileservers

Uses for Automation

Scripts can be used for automating specific tasks. Automation is used to replace a repetitive manual step with one that happens automatically. Humans are fallible. They can become tired, make mistakes, fail to follow instructions, be inconsistent in their job performance, and more. In contrast, automated processes complete instructions exactly as coded, in a consistent manner. They can run 24 hours a day, everyday, without tiring. For many tasks that are appropriate for automation, it can be more cost effective to use automation than human labor.

Appropriate uses for automation include:

- The automatic timing and regulation of traffic lights
- A repetitive task that is at high risk for human error
- Sending commands to a computer
- Detecting and removing duplicates of data
- Sending automated emails that are personalized by pulling individual names from a database and plugging them into the email
- Updating a large number of file permissions
- Reporting on system data, like disk or memory usage
- Installing software
- Generating reports
- Deploying a file or a computer program to all computers on a company network
- Using a configuration management system to deploy software patches, after a human has *designed*

the system

- Populating an e-commerce site with products
- Setting the home directory and access permissions for users

Automation is not always an appropriate or complete solution

Automation cannot perform all human work. Tasks that call for human creativity, social connection, psychology, flexibility, ingenuity, evaluation, and/or complex analytic work are not good candidates for full automation. Sometimes automation can be used to perform one or more subtasks of a larger set of tasks – but – human intervention is required to complete the tasks. The following are some examples of tasks that cannot or should not be **fully** automated:

- Items that require human evaluation and analytic skills:
    - *Designing* a configuration management system
    - Investigating and troubleshooting all end user problems
    - Writing a computer program
    - Building a new startup business
- Items that require human creativity and/or an eye for aesthetic qualities*:*
    - Designing an attractive webpage *(AI can do this, but simple automation cannot)*
    - Wedding photography
    - Haircuts and styling
- Items that cannot be automated due to basic physics:
    - Troubleshooting or repairing machines that cannot power on or boot up
- Items that need human interaction, psychology, and/or evaluation skills:
    - Interviewing and hiring new employees
    - Customer service *(chat bots cannot address every customer service need)*
- Items that should not be fully automated due to costs and safety:
    - Grocery store checkout process, including bagging groceries
    - Tasks that are less expensive to perform manually

Artificial Intelligence

It is important to understand that basic automation is not the same as artificial intelligence. Automation is used to explicitly instruct a machine on how to perform a task. Artificial intelligence (AI) involves training a computing machine to perform more complex tasks through a process called machine learning. This process prepares the AI software to perform new tasks without a human needing to program explicit instructions for each task. Although AI is often used for automating human tasks, AI automation is much more complex than basic automation.

From <https://www.coursera.org/learn/python-crash-course/supplement/VilN1/uses-for-automation>

Practice Quiz: Introduction to Programming

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/5nR8G/practice-quiz-introduction-to-programming

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M1%20Hello%20Python!/practice-quiz-intro-to-programming.md>

What is python?

```
friends = ['Taylor', 'Alex', 'Pat', 'Eli']

for friend in friends:
    print("Hi" + friend + ',')
```

➢ HiTaylor,
HiAlex,
HiPat,
HiEli,

A Note on Syntax and Code Blocks

When writing code, using correct syntax is super important. Even a small typo, like a missing parentheses or an extra comma, can cause a syntax error and the code won't execute at all. Yikes. If your code results in an error or an exception, pay close attention to syntax and watch out for minor mistakes.

If your syntax is correct, but the script has **unexpected behavior** or output, this may be due to a **semantic** problem. Remember that syntax is the rules of how code is constructed, while semantics are the overall effect the code has. It is possible to have syntactically correct code that runs successfully, but doesn't do what we want it to do.

When working with the code blocks in exercises for this course, be mindful of syntax errors, along with the overall result of your code. Just because you fixed a syntax error doesn't mean that the code will have the desired effect when it runs! Once you've fixed an error in your code, don't forget to submit it to have your work checked.

From <https://www.coursera.org/learn/python-crash-course/supplement/56Fm5/a-note-on-syntax-and-code-blocks>

Why is Python relevant to today's IT industry?

- Python scripts are easy to write, understand, and maintain.
- There are many system administration tools built with Python.
- Python is available on a wide variety of platforms.

```
for i in range(10):
  print("Hello, World!")
```

➢ Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!

Practice Quiz: Introduction to Python

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/2VOEJ/practice-quiz-introduction-to-python

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M1%20Hello%20Python!/practice-quiz-intro-to-python.md#question-4>

3. Hello world

```
print ("I'm programming in Python!")
```

- I'm programming in python

```
color = "Red"
thing = "Love"
print(color + " is the color of " + thing)
```

- Red is the color of Love

```
print((((1+2)*3)/4)**5)
```

- 57.6650390625

First Programming Concepts Cheat Sheet

Functions and Keywords

Functions and keywords are the building blocks of a language's syntax.

Functions are pieces of code that perform a unit of work. In the examples we've seen so far, we've only encountered the print() function, which prints a message to the screen. We'll learn about a lot of other functions in later lessons but, if you're too curious to wait until then, you can discover all the functions available here.

Keywords are reserved words that are used to construct instructions. We briefly encountered for and in in our first Python example, and we'll use a bunch of other keywords as we go through the course. For reference, these are all the reserved keywords:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

You don't need to learn this list; we'll dive into each keyword as we encounter them. In the meantime, you can see examples of keyword usage here.

Arithmetic operators

Python can operate with numbers using the usual mathematical operators, and some special operators, too. These are all of them (we'll explore the last two in later videos).

- **a + b** = Adds a and b
- **a - b** = Subtracts b from a
- **a * b** = Multiplies a and b
- **a / b** = Divides a by b
- **a ** b** = Elevates a to the power of b. For non-integer values of b, this becomes a root (i.e. a**(1/2) is the square root of a)
- **a // b** = The integer part of the integer division of a by b
- **a % b** = The remainder part of the integer division of a by b

From <https://www.coursera.org/learn/python-crash-course/supplement/nonTo/first-programming-concepts-cheat-sheet>

Practice Quiz: Hello World

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/q9yr3/practice-quiz-hello-world

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M1%20Hello%20Python!/practice-quiz-hello-world.md>

Graded assessment

Assessment:

https://www.coursera.org/learn/python-crash-course/exam/Zcevo/module-1-graded-assessmenthttps://www.coursera.org/learn/python-crash-course/exam/Zcevo/module-1-graded-assessment

Module 1 Graded Assessment Solution

From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M1%20Hello%20Python!/graded-assessments/quiz-module-1-graded-assessments.md>

1. Expressions and variables

- Why does this code raise an error:

```
print("1234"+5678)
```

Error on line 1:
print("1234"+5678)
TypeError: must be str, not int

- Because Python doesn't know how to add a number to a string.
- Search for errors in your favorite search engine (google for example) OR use the type() function:

```
print(type("a"))
 ➢ <class 'str'>
print(type(2))
 ➢ <class 'int'>
print(type(2.5))
 ➢ <class 'float'>
```

Data Types Recap

In Python, text in between quotes -- either single or double quotes -- is a string data type. An integer is a whole number, without a fraction, while a float is a real number that can contain a fractional part. For example, 1, 7, 342 are all integers, while 5.3, 3.14159 and 6.0 are all floats. When attempting to mix incompatible data types, you may encounter a **TypeError**. You can always check the data type of something using the *type()* function.

From <https://www.coursera.org/learn/python-crash-course/supplement/IPkxV/data-types-recap>

```
print(7+8.5)
 ➢ 15.5
```

Implicit conversion:
The interpreter automatically converts one data type into another.
- Interpreter converted integer 7 into a float 7.

```
base = 5
height = 3
area = (base*height)/2
print(area)
 ➢ 7.5
```

Variable Restrictions: (Case sensitive)
- Don't use keywords or reserved words
- Don't use spaces
- Must start with a letter or underscore (_)
- Must be made up of only letters, numbers, and underscores(_)

```
print("The area of the triangle is: "+str(area))
 ➢ The area of the triangle is: 7.5
```

Explicit conversion: e.g.: To combine a string and a number

To convert one data type and another, we call a function with the name of the type we are converting to.

```
total = 2048 + 4357 + 97658 + 125 + 8
files = 5
average = total / files
print("The average size is: " + str(average))
```

- The average size is: 20839.2

Implicit vs Explicit Conversion

As we saw earlier in the video, some data types can be mixed and matched due to implicit conversion. ***Implicit*** conversion is where the interpreter helps us out and ***automatically*** converts one data type into another, without having to explicitly tell it to do so.

By contrast, ***explicit*** conversion is where we ***manually*** convert from one data type to another by calling the relevant function for the data type we want to convert to. We used this in our video example when we wanted to print a number alongside some text. Before we could do that, we needed to call the ***str()*** function to convert the number into a string. Once the number was explicitly converted to a string, we could join it with the rest of our textual string and print the result.

From <https://www.coursera.org/learn/python-crash-course/supplement/kzyZn/implicit-vs-explicit-conversion>

Practice Quiz: Expressions and Variables

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/yMizb/practice-quiz-expressions-and-variables/attempt

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M2%20Basic%20Python%20Syntax/practice-quiz-expressions-and-variables.md>

Question 2: Integer Division

```
numerator = 10
denominator = 10
result = numerator // denominator
print(result)
```

- 1

```
numerator = 10
denominator = 10
result = numerator / denominator
print(result)
```

- 1.0

Question 3: for loop inside the print function

```
word1 = "How"
```

```
word2 = "do"
word3 = "you"
word4 = "like"
word5 = "Python"
word6 = "so"
word7 = "far?"

print(' '.join([eval("word"+str(i+1)) for i in range(7)]))
```

- How do you like python so far?

#The eval() function takes the **string** and evaluate it as a **Python expression**
#The join() method returns a string created by joining the elements of an iterable by the given string separator. Some of the example of iterables are: List[word1 word2 ..], Tuple, String, Dictionary and Set.

Defining Functions

```
def print_seconds(hours, minutes, seconds):
    print(seconds+minutes*60+hours*3600)

print_seconds(1,2,3)
```

- 3723

Defining Functions Recap

We looked at a few examples of built-in functions in Python, but being able to define your own functions is incredibly powerful. We start a function definition with the def keyword, followed by the name we want to give our function. After the name, we have the parameters, also called arguments, for the function enclosed in parentheses. A function can have no parameters, or it can have multiple parameters. Parameters allow us to call a function and pass it data, with the data being available inside the function as variables with the same name as the parameters. Lastly, we put a colon at the end of the line.

After the colon, the function body starts. It's important to note that in Python the function body is delimited by indentation. This means that all code indented to the right following a function definition is part of the function body. The first line that's no longer indented is the boundary of the function body. It's up to you how many spaces you use when indenting -- just make sure to be consistent. So if you choose to indent with four spaces, you need to use four spaces everywhere in your code.

From <https://www.coursera.org/learn/python-crash-course/supplement/0jefW/defining-functions-recap>

```
def get_seconds(hours, minutes, seconds):
  return 3600*hours + 60*minutes + seconds

amount_a = get_seconds(2,30,0)
amount_b = get_seconds(0,45,15)
result = amount_a+amount_b

print("The result is: " + str(result))
```

- The result is: 11715

Another Way

```
def convert_seconds(seconds):
    hours=seconds//3600
    minutes=(seconds%3600)//60
    remaining_seconds=(seconds%3600)%60
    return hours, minutes, remaining_seconds

hours, minutes, remaining_seconds= convert_seconds(5000)
print(hours, minutes, remaining_seconds)
```

```
def convert_seconds(seconds):
    hours= seconds//3600
    #minutes=int((seconds/3600-hours)*60)
    minutes=(seconds-hours*3600)//60
    remaining_seconds=seconds-hours*3600- minutes*60
    return hours, minutes, remaining_seconds

hours, minutes, remaining_seconds= convert_seconds(5000)
print(hours, minutes, remaining_seconds)
```

- 1, 23, 20

```
def greeting(name):
    print("Welcome, " + name)

result = greeting("Christine")
print(result)
```

- Welcome, Christine
- None

None

A very special data type in Python used to indicate that things are empty or that they return nothing.

Returning Values Using Functions

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy. We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

From <https://www.coursera.org/learn/python-crash-course/supplement/idPEm/returning-values-using-functions>

```
def month_days(month, days):
    print(month + " has " + str(days) + " days.")

month_days("June", 30)
month_days("July", 31)
```

➢ June has 30 days.
July has 31 days.

Self-documenting code

It is written in a way that's readable and doesn't conceal its intent.

Refactoring

In programming lingo, when we re-write code to be more self-documenting, we call this process refactoring.

Before refactoring:

```
def f1(x, y):
    z = x*y  # the area is base*height
    print("The area is " + str(z))
```

After refactoring:

```
def rectangle_area(base, height):
    area = base*height  # the area is base*height
    print("The area is " + str(area))
rectangle_area(5, 6)
```

This is how you write a comment in python

Practice Quiz: Functions

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/hqfxs/practice-quiz-functions?redirectToCover=true

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M2%20Basic%20Python%20Syntax/practice-quiz-functions.md>

Question 4: Integer Division

- Let's revisit our lucky_number function. We want to change it, so that instead of printing the message, it returns the message:

```
def lucky_number(name):
  number = len(name) * 9
  message = "Hello " + name + ". Your lucky number is " + str(number)
  return message

print(lucky_number("Kay"))
print(lucky_number("Cameron"))
```

➢ Hello Kay. Your lucky number is 27
Hello Cameron. Your lucky number is 63

3. Conditionals

Saturday, May 15, 2021 11:41 PM

Boolean

One of two possible states: either true or false.

```
>>> print(10>1)
True
>>> print("cat" == "dog")
False
>>> print(1 != 2)
True
>>> print(1 < "1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
>>> print( 1 == "1")
False
>>>
```

```
print("cat" > "Cat")   # In Python uppercase letters are alphabetically sorted before lowercase letters.
```

➢ True

Logical operators

- To evaluate as true, the **and** operator would need both expressions to be true at the same time.
- If we use the **or** operator, instead, the expression will be true if either of the expressions are true, and false only when both expressions are false.
- The **not** operator inverts the value of the expression that's in front of it.

```
>>> print("Yellow" > "Cyan" and "Brown" > "Magenta")
False
>>> print(25 > 50 or 1 != 2)
True
>>> print(not 42 == "Answer")
True
>>>
```

Comparison Operators

In Python, we can use comparison operators to compare values. When a comparison is made, Python returns a boolean result, or simply a True or False.

- To check if two values are the same, we can use the equality operator: **==**
- To check if two values are not the same, we can use the not equals operator: **!=**

We can also check if values are greater than or lesser than each other using **>** and **<**. If you try to compare data types that aren't compatible, like checking if a string is greater than an integer, Python will throw a **TypeError**.

We can make very complex comparisons by joining statements together using logical operators with our comparison operators. These logical operators are **and**, **or**, and **not**. When using the **and** operator, both sides of the statement being evaluated must be true for the whole statement to be true. When using the **or** operator, if either side of the comparison is true, then the whole statement is true. Lastly, the **not** operator simply inverts the value of the statement immediately following it. So if a statement evaluates to True, and we put the **not** operator in front of it, it would become False.

From <https://www.coursera.org/learn/python-crash-course/supplement/2d3nc/comparison-operators>

Branching

The ability of a program to alter its execution sequence.

- The body of the if block will only execute when the condition evaluates to true; otherwise it's skipped.

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
```

- The is_positive function should return True if the number received is positive, otherwise it returns None:

```
def is_positive(number):
  if number>0:
    return True
```

if Statements Recap

We can use the concept of **branching** to have our code alter its execution sequence depending on the values of variables. We can use an *if* statement to evaluate a comparison. We start with the *if* keyword, followed by our comparison. We end the line with a colon. The body of the *if* statement is then indented to the right. If the comparison is ***True***, the code inside the *if* body is executed. If the comparison evaluates to ***False***, then the code block is skipped and will not be run.

From <https://www.coursera.org/learn/python-crash-course/supplement/v9Us9/if-statements-recap>

else Statements

```
def is_positive(number):
  if number > 0:
    return True
  else:
    return False


def is_even(number):
    if number % 2 == 0:
        return True
    return False
```

else Statements and the Modulo Operator

We just covered the if statement, which executes code if an evaluation is true and skips the code if it's false. But what if we wanted the code to do something different if the evaluation is false? We can do this using the else statement. The else statement follows an if block, and is composed of the keyword else followed by a colon. The body of the else statement is indented to the right, and will be executed if the above if statement doesn't execute.

We also touched on the modulo operator, which is represented by the percent sign: %. This operator performs integer division, but only returns the remainder of this division operation. If we're dividing 5 by 2, the quotient is 2, and the remainder is 1. Two 2s can go into 5, leaving 1 left over. So 5%2 would return 1. Dividing 10 by 5 would give us a quotient of 2 with no remainder, since 5 can go into 10 twice with nothing left over. In this case, 10%2 would return 0, as there is no remainder.

elif Statements

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    elif len(username) > 15:
        print("Invalid username. Must be at most 15 characters long")
    else:
        print("Valid username")
```

Building off of the *if* and *else* blocks, which allow us to branch our code depending on the evaluation of one statement, the *elif* statement allows us even more comparisons to perform more complex branching. Very similar to the *if* statements, an *elif* statement starts with the *elif* keyword, followed by a comparison to be evaluated. This is followed by a colon, and then the code block on the next line, indented to the right. An *elif* statement must follow an *if* statement, and will only be evaluated if the *if* statement was evaluated as false. You can include multiple *elif* statements to build complex branching in your code to do all kinds of powerful things!

From <https://www.coursera.org/learn/python-crash-course/supplement/BHcFC/more-complex-branching-with-elif-statements>

Conditionals Cheat Sheet

Conditionals Cheat Sheet

In earlier videos, we took a look at some of the built-in Python operators that allow us to compare values, and some logical operators we can use to combine values. We also learned how to use operators in if-else-elif blocks.

It's a lot to learn but, with practice, it gets easier to remember it all. In the meantime, this handy cheat sheet gives you all the information you need at a glance.

Comparison operators

- a == b: a is equal to b
- a != b: a is different than b
- a < b: a is smaller than b
- a <= b: a is smaller or equal to b
- a > b: a is bigger than b
- a >= b: a is bigger or equal to b

Logical operators

- a and b: True if both a and b are True. False otherwise.
- a or b: True if either a or b or both are True. False if both are False.
- not a: True if a is False, False if a is True.

Branching blocks

In Python, we branch our code using if, else and elif. This is the branching syntax:

```
if condition1:
    if-block
elif condition2:
    elif-block
else:
    else-block
```

Remember: The if-block will be executed if condition1 is True. The elif-block will be executed if condition1 is False and condition2 is True. The else block will be executed when all the specified conditions are false.

From <https://www.coursera.org/learn/python-crash-course/supplement/R9diu/conditionals-cheat-sheet>

Practice Quiz: Functions

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/QrKqf/practice-quiz-conditionals/attempt

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M2%20Basic%20Python%20Syntax/practice-quiz-conditionals.md>

```
def calculate_storage(filesize):
    block_size = 4096
    # Use floor division to calculate how many blocks are fully occupied
    full_blocks = filesize//block_size
    # Use the modulo operator to check whether there's any remainder
    partial_block_remainder = filesize%block_size
    # Depending on whether there's a remainder or not, return
    # the total number of bytes required to allocate enough blocks
    # to store your data.
    if partial_block_remainder > 0:
        return (full_blocks+1)*block_size
    return full_blocks*block_size
print(calculate_storage(1))    # Should be 4096
print(calculate_storage(4096)) # Should be 4096
print(calculate_storage(4097)) # Should be 8192
print(calculate_storage(6000)) # Should be 8192
```

While Loops

Wednesday, February 22, 2023 3:35 AM

```
x=0
while x < 5:
    print("Not there yet, x=" + str(x))
    x = x + 1
print("x=" + str(x))
```

```
>>> x = 0
>>> while x < 5:
...     print("Not there yet, x=" + str(x))
...     x = x + 1
...
...
Not there yet, x=0
Not there yet, x=1
Not there yet, x=2
Not there yet, x=3
Not there yet, x=4
>>> print("x=" + str(x))
x=5
```

Anatomy of a While Loop

A *while* loop will continuously execute code depending on the value of a condition. It begins with the keyword *while*, followed by a comparison to be evaluated, then a colon. On the next line is the code block to be executed, indented to the right. Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true. What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true. Once the statement is no longer true, the loop exits and the next line of code will be executed.

From <https://www.coursera.org/learn/python-crash-course/supplement/3ot5p/anatomy-of-a-while-loop>

```
def attempts(n):
    x = 1
    while x <= n:
        print("Attempt " + str(x))
        x += 1
    print("Done")

attempts(5)
```

```
username = get_username()
while not valid_username(username):
    print("Invalid username")
    username = get_username()
```

Why Initializing Variables Matters

```
>>> while my_variable < 10:
...   print("Hello")
...   my_variable +=1
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_variable' is not defined
>>>
```

NameError

```
def count_down(start_number):
  current=start_number
  while (current > 0):
    print(current)
```

```
    current -= 1
  print("Zero!")
count_down(3)
```

Common Pitfalls With Variable Initialization

You'll want to watch out for a common mistake: forgetting to initialize variables. If you try to use a variable without first initializing it, you'll run into a **NameError**. This is the Python interpreter catching the mistake and telling you that you're using an undefined variable. The fix is pretty simple: initialize the variable by assigning the variable a value before you use it.

Another common mistake to watch out for that can be a little trickier to spot is forgetting to initialize variables with the correct value. If you use a variable earlier in your code and then reuse it later in a loop without first setting the value to something you want, your code may wind up doing something you didn't expect. Don't forget to initialize your variables before using them!

From <https://www.coursera.org/learn/python-crash-course/supplement/Xupnx/common-pitfalls-with-variable-initialization>

```
x=64
while x != 0 and x % 2 == 0:
    x = x /2
    print(x)
```

```
while True:
    do_something_cool()
    if user_requested_to_stop():
        break
#break is used to exit the loop
```

Infinite loops and Code Blocks

Another easy mistake that can happen when using loops is introducing an infinite loop. An infinite loop means the code block in the loop will continue to execute and never stop. This can happen when the condition being evaluated in a *while* loop doesn't change. Pay close attention to your variables and what possible values they can take. Think about unexpected values, like zero.

In the Coursera code blocks, you may see an error message that reads "Evaluation took more than 5 seconds to complete." This means that the code encountered an infinite loop, and it timed out after 5 seconds. You should take a closer look at the code and variables to spot where the infinite loop is.

From <https://www.coursera.org/learn/python-crash-course/supplement/k4CY9/infinite-loops-and-code-blocks>

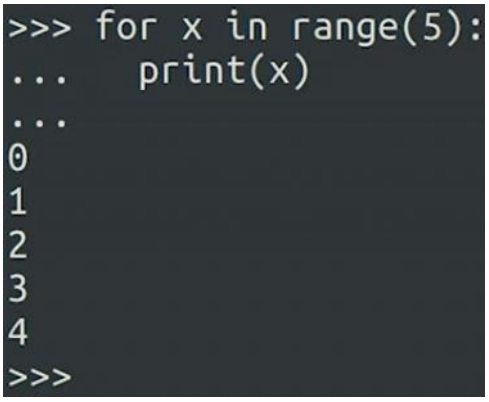
Practice Quiz: Functions

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/64Zai/practice-quiz-while-loops/attempt

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M3%20Loops/practice-quiz-while-loops.md>

for loops

Thursday, February 23, 2023 3:45 AM




```
values = [23, 52, 59, 37, 48]
sum = 0
length = 0
for value in values:
    sum +=value
    length += 1
print("Total sum: "+ str(sum) + " - Average: " + str(sum/length))
```

For Loops Recap

For loops allow you to iterate over a sequence of values. Let's take the example from the beginning of the video:

for x in range(5):

  print(x)

Similar to *if* statements and *while* loops, *for* loops begin with the keyword ***for*** with a colon at the end of the line. Just like in function definitions, *while* loops and *if* statements, the body of the *for* loop begins on the next line and is indented to the right. But what about the stuff in between the *for* keyword and the colon? In our example, we're using the *range()* function to create a sequence of numbers that our *for* loop can iterate over. In this case, our variable **x** points to the current element in the sequence as the *for* loop iterates over the sequence of numbers. Keep in mind that in Python and many programming languages, a range of numbers will start at 0, and the list of numbers generated will be one less than the provided value. So *range(5)* will generate a sequence of numbers from 0 to 4, for a total of 5 numbers.

Bringing this all together, the range(5) function will create a sequence of numbers from 0 to 4. Our *for* loop will iterate over this sequence of numbers, one at a time, making the numbers accessible via the variable **x** and the code within our loop body will execute for each iteration through the sequence. So for the first loop, **x** will contain 0, the next loop, 1, and so on until it reaches 4. Once the end of the sequence comes up, the loop will exit and the code will continue.

The power of *for* loops comes from the fact that it can iterate over a sequence of any kind of data, not just a range of numbers. You can use *for* loops to iterate over a list of strings, such as usernames or lines in a file.

Not sure whether to use a *for* loop or a *while* loop? Remember that a *while* loop is great for performing an action over and over until a condition has changed. A *for* loop works well when you want to iterate over a sequence of elements.

From <https://www.coursera.org/learn/python-crash-course/supplement/FCEnY/for-loops-recap>

```
>>> product = 1
>>> for n in range(1,10):
...     product = product * n
...
>>> print(product)
362880
```

```
for x in range(2, -2, -1):
    print(x)
```

➢
```
2
1
0
-1
```

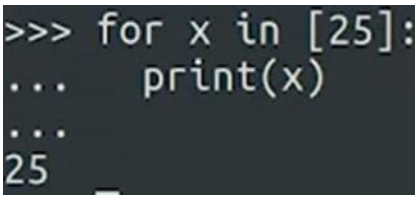
```
for left in range(7):
    for right in range(left,7):
        print("[" + str(left) + "|" + str(right)+ "]", end=" ")
    print()
```

➢
```
[0|0] [0|1] [0|2] [0|3] [0|4] [0|5] [0|6]
[1|1] [1|2] [1|3] [1|4] [1|5] [1|6]
[2|2] [2|3] [2|4] [2|5] [2|6]
[3|3] [3|4] [3|5] [3|6]
[4|4] [4|5] [4|6]
[5|5] [5|6]
[6|6]
```

```
teams = ['Dragon', 'Wolves', 'Pandas', 'Unicorns']
for home_team in teams:
    for away_team in teams:
        if home_team != away_team:
            print(home_team + " vs "  + away_team)
```

➢
```
Dragon vs Wolves
Dragon vs Pandas
Dragon vs Unicorns
Wolves vs Dragon
Wolves vs Pandas
Wolves vs Unicorns
Pandas vs Dragon
Pandas vs Wolves
Pandas vs Unicorns
Unicorns vs Dragon
Unicorns vs Wolves
Unicorns vs Pandas
```

```
>>> for x in 25:
...     print(x)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```




```
def greet_friends(friends):
    for friend in friends:
        print("Hi " + friend)
greet_friends(['Taylor', 'Luisa', 'Jamaal', 'Eli'])
```

- Hi Taylor
  Hi Luisa
  Hi Jamaal
  Hi Eli
  Hi Taylor
  Hi Luisa
  Hi Jamaal
  Hi Eli

```
greet_friends("Barry")
```

- Hi B
  Hi a
  Hi r
  Hi r
  Hi y

Practice Quiz: Functions

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/64Zai/practice-quiz-while-loops/

Solution:
From <https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M3%20Loops/practice-quiz-for-loops.md>

3. Which for loops will print all even numbers from 0 to 18? Select all that apply.

```
for n in range(19):
    if n % 2 == 0:
        print(n)
```

```
for n in range(10):
    print(n+n)
```

5. Write a for loop with a three parameter range() function that prints the multiples of 7 between 0 and 100.

```
for i in range(0,100+1,7):
    print(i)
```

From <https://www.coursera.org/learn/python-crash-course/quiz/C9MVo/practice-quiz-for-loops/attempt>

Recursion

Friday, June 30, 2023 5:33 PM

Recursion:

The repeated application of the same procedure to a smaller problem

Recursion let us tackle complex problems by reducing the problem to a simpler one

Nested dolls:

How many dolls

Line and how many people in front of you:

You ask the person in front of you and they ask the same question in front of them until the first person, then backwards count+1

In programming, recursion is a way of doing a repetitive task by having a function call itself.

Until it reaches the condition that is called the **Base Case**

```
def factorial(n):
    if n < 2: # base case
        return 1
    return n*factorial(n-1) #recursive case

factorial(5)
```

```
def factorial(n):
    print('Factoral called with ' + str(n))
    if n < 2: # base case
        print('Returning 1')
        return 1
    result=n*factorial(n-1)
    print('Returning ' + str(result) + ' for factorial of ' + str(n))
    return result

factorial(4)
```

```
>>>
Factoral called with 4
Factoral called with 3
Factoral called with 2
Factoral called with 1
Returning 1
Returning 2 for factorial of 2
Returning 6 for factorial of 3
Returning 24 for factorial of 4
>>>
```

```
def sum_positive_numbers(n):
    # The base case is n being smaller than 2
    if n < 2:
        return 1
    # The recursive case is adding this number to
    # the sum of the numbers smaller than this one.
    return n + sum_positive_numbers(n-1)
print(sum_positive_numbers(3)) # Should be 6
print(sum_positive_numbers(5)) # Should be 15
```

Not only math functions, but also in recursive structures, for example:

1) To calculate the number of files inside a directory
    a. Base case is the number of files in a subdirectory that has no subdirectory
    b. Recursive case is the number of files in a subdirectory that has subdirectory
2) Groups that include other groups in active directory and LDAP

```
def factorial(n):
    if n < 2: # base case
        return 1
    return n*factorial(n-1) #recursive case

factorial(1000)
```

```
>>>
Error on line 6:
  factorial(1000)
Error on line 4:
  return n*factorial(n-1) #recursive case
Error on line 4:
  return n*factorial(n-1) #recursive case
Error on line 4:
  return n*factorial(n-1) #recursive case
 [Previous line repeated 987 more times]
Error on line 2:
  if n < 2: # base case
RecursionError: maximum recursion depth exceeded in comparison
>>>
```

Valid only when the recursive structure won't reach a thousand nested levels

Practice Quiz: Recursion

Quiz:
https://www.coursera.org/learn/python-crash-course/quiz/64Zai/practice-quiz-while-loops/

Solution:
https://github.com/AayushTyagi1/google-it-automation/blob/master/C1%20Crash%20Course%20on%20Python/M3%20Loops/practice-quiz-recursion.md

Quiz:

```
def is_power_of(number, base):
  # Base case: when number is smaller than base.
  if number < base:
    # If number is equal to 1, it's a power (base**0).
    if number == 1:
      return True
  # Recursive case: keep dividing number by base.
  #if the number not divisible by the base then return False
  if number%base != 0:
    return False
  else:
    return is_power_of(number=number//base, base=base)
```

```
print(is_power_of(8,2)) # Should be True
print(is_power_of(64,4)) # Should be True
print(is_power_of(70,10)) # Should be False
```

```
def count_users(group):
  count = 0
  for member in get_members(group):
    if is_group(member):
      count += count_users(member)
    else:
      count += 1
  return count

print(count_users("sales")) # Should be 3
print(count_users("engineering")) # Should be 8
print(count_users("everyone")) # Should be 18
```

```
>>> name = "Jaylen"
>>> print(name[1])
a
>>> print(name[0])
J
>>> print(name[5])
n
>>> print(name[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

```
>>> text = "Random string with a lot of characters"
>>> print(text[-1])
s
>>> print(text[-2])
r
>>>
```

```
>>> color = "Orange"
>>> color[1:4]
'ran'
>>>
```

```
>>> fruit = "Pineapple"
>>> print(fruit[:4])
Pine
>>> print(fruit[4:])
apple
>>>
```

String Indexing and Slicing

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that's larger than the length of your string, you'll get an **IndexError**. This is because you're trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5].

This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. For example:

>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang'

The slice *includes* the character at index 1, and *excludes* the character at index 4. You can also easily reference a substring at the start or end of the string by only specifying one end of the range. For example, only giving the end of the range:

>>> fruit[:5] 'Mango'

This gave us the characters from the start of the string through index 4, *excluding* index 5. On the other hand this example gives is the characters *including* index 5, through the end of the string:

>>> fruit[5:] 'steen'

You might have noticed that if you put both of those results together, you get the original string back!

>>> fruit[:5] + fruit[5:] 'Mangosteen'

Cool!

```
>>> message = "A kong string with a silly typo"
>>> message[2] = "l"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> new_message = message[0:2] + "l" + message[3:]
>>> print(new_message)
A long string with a silly typo
>>> message = "This is a new message"
>>> print(message)
This is a new message
>>> message = "And another one"
>>> print(message)
And another one
>>>
```

```
>>> pets = "Cats & Dogs"
>>> pets.index("&")
5
>>> pets.index("C")
0
>>> pets.index("Dog")
4
>>> pets.index("s")
3
>>> pets.index("x")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

```
>>> pets = "Cats & Dogs"
>>> pets.index("&")
5
>>> pets.index("C")
0
>>> pets.index("Dog")
4
>>> pets.index("s")
3
>>> pets.index("x")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "Dragons" in pets
False
>>> "Cats" in pets
True
>>>
```

```
def replace_domain(email, old_domain, new_domain):
    if "@" + old_domain in email:
        index=email.index("@" + old_domain)
        new_email = email[:index] + "@" + new_domain
        return new_email
    return email
```

Basic String Methods

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that's the start of the substring. If there's more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we'll receive a **ValueError** explaining that the substring was not found.

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses" in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren't included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

```
>>> "Mountains".upper()
'MOUNTAINS'
>>> "Mountains".lower()
'mountains'
>>>
```

```
>>> answer = 'YES'
>>> if answer.lower() == "yes":
...   print("User said yes")
...
User said yes
>>>
```

```
>>> " yes ".strip()
'yes'
>>> " yes ".lstrip()
'yes '
>>> " yes ".rstrip()
' yes'
>>>
```

```
>>> "The number of times e occurs in this string is 4".count("e")
4
>>>
```

```
>>> "Forest".endswith("rest")
True
>>>
```

```
>>> "Forest".isnumeric()
False
>>> "12345".isnumeric()
True
>>> int("12345") + int("54321")
66666
>>>
```

```
>>> " ".join(["This", "is", "a", "phrase", "joined", "by", "spaces"])
'This is a phrase joined by spaces'
>>> "...".join(["This", "is", "a", "phrase", "joined", "by", "triple", "dots"])
'This...is...a...phrase...joined...by...triple...dots'
>>>
```

```
>>> "This is another example".split()
['This', 'is', 'another', 'example']
>>>
```

Advanced String Methods

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

```
>>> name = "Manny"
>>> number = len(name) * 3
>>> print("Hello {}, your lucky number is {}".format(name, number))
Hello Manny, your lucky number is 15
>>> print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)*3))
Your lucky number is 15, Manny.
>>>
```

```
>>> price = 7.5
>>> with_tax = price * 1.09
>>> print(price, with_tax)
7.5 8.175
>>> print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with_tax))
Base price: $7.50. With Tax: $8.18
>>>
```

```
>>> def to_celsius(x):
...     return (x-32)*5/9
...
>>> for x in range(0,101,10):
...     print(x, to_celsius(x))
...
0 -17.77777777777778
10 -12.222222222222221
20 -6.666666666666667
30 -1.1111111111111112
40 4.444444444444445
50 10.0
60 15.555555555555555
70 21.11111111111111
80 26.666666666666668
90 32.22222222222222
100 37.77777777777778
>>>
```

```
>>> def to_celsius(x):
...     return (x-32)*5/9
...
>>> for x in range(0,101,10):
...     print("{:>3} F | {:>6.2f} C".format(x, to_celsius(x)))
...
  0 F | -17.78 C
 10 F | -12.22 C
 20 F |  -6.67 C
 30 F |  -1.11 C
 40 F |   4.44 C
 50 F |  10.00 C
 60 F |  15.56 C
 70 F |  21.11 C
 80 F |  26.67 C
 90 F |  32.22 C
100 F |  37.78 C
>>>
```

String Formatting

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they'll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you'd format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

List

Friday, February 24, 2023 5:37 AM

```
>>> x = ["Now", "we", "are", "cooking!"]
>>> type(x)
<class 'list'>
>>> print(x)
['Now', 'we', 'are', 'cooking!']
>>> len(x)
4
>>> "are" in x
True
>>> "Today" in x
False
>>> print(x[0])
Now
>>> print(x[3])
cooking!
>>> print(x[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

```
>>> x[1:3]
['we', 'are']
>>> x[:2]
['Now', 'we']
>>> x[2:]
['are', 'cooking!']
```

```
def get_word(sentence, n):
    # Only proceed if n is positive
    if n > 0:
        words = sentence.split()
        # Only proceed if n is not more than the number of words
        if n <= len(words):
            return(words[n-1])
    return("")
print(get_word("This is a lesson about lists", 4)) # Should print: lesson
print(get_word("This is a lesson about lists", -4)) # Nothing
print(get_word("Now we are cooking!", 1)) # Should print: Now
print(get_word("Now we are cooking!", 5)) # Nothing
```

Lists Defined

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: **list = ["This", "is", "a", "list"]**. You can use the **len()** function to return the number of elements in a list: **len(list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the eleme nt is not found in the list, it will return False. For example, **"This" in list** would return True in our example. Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You c an access the first element in a list by doing **list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They're both examples of sequences of data. Sequences have similar properties , like (1) being able to iterate over them using **for loops**; (2) support indexing; (3) using the **len** function to find the length of the sequence; (4) using the plus operator **+** in order to concatenate; and (5) using the **in** keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence ty pes as well.

```
>>> fruits = ["Pineapple", "Banana", "Apple", "Melon"]
>>> fruits.append("Kiwi")
>>> print(fruits)
['Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi']
>>> fruits.insert(0, "Orange")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi']
>>> fruits.insert(25, "Peach")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Melon', 'Kiwi', 'Peach']
>>> fruits.remove("Melon")
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Apple', 'Kiwi', 'Peach']
>>> fruits.remove("Pear")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> fruits.pop(3)
'Apple'
>>> print(fruits)
['Orange', 'Pineapple', 'Banana', 'Kiwi', 'Peach']
>>> fruits[2] = "Strawberry"
>>> print(fruits)
['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach']
>>>
```

Modifying Lists

While lists and strings are both sequences, a big difference between them is that lists are mutable. This means that the cont ents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the method **insert**. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to th e list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the li st. It would just shift all the other elements by one. If you specify an index that's larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn't fo und in the list, you'll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it's located. This can also be useful when you need to access the data and also want to remove i t from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For exam ple, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

Strings: " "
Sequences of characters, and are immutable
Lists: [ ]
Sequences of elements of any type, and are mutable
Tuples: ()
Sequences of elements of any type that are immutable

We want to make sure an element in a certain position refers to specific thing and won't change

```
fullname=(Abdelrahman, Ahmed, Samy)
```

The positions of the elements inside the tuple have meaning

Uses:
1) return value of tuple of 3 elements for example:

```
>>> def convert_seconds(seconds):
...     hours = seconds // 3600
...     minutes = (seconds - hours * 3600) // 60
...     remaining_seconds = seconds - hours * 3600 - minutes * 60
...     return hours, minutes, remaining_seconds
...
>>> result = convert_seconds(5000)
>>> type(result)
<class 'tuple'>
>>> print(result)
(1, 23, 20)
>>>
```

2) We can Unpack them:
3 elements becomes 3 separate variables:

```
>>> print(result)
(1, 23, 20)
>>> hours, minutes, seconds = result
>>> print(hours, minutes, seconds)
1 23 20
>>> hours, minutes, seconds = convert_seconds(1000)
>>> print(hours, minutes, seconds)
0 16 40
>>>
```

3) It's common to use tuple to represent data that has more than 1 value that needs to be kept together
Filename and its size
Name and email
Time and System health at that time

```
def file_size(file_info):
    name, itstype, size_Byte= file_info
    return("{:.2f}".format(size_Byte / 1024))

print(file_size(('Class Assignment', 'docx', 17875))) # Should print 17.46
print(file_size(('Notes', 'txt', 496))) # Should print 0.48
print(file_size(('Program', 'py', 1239))) # Should print 1.21
```

Tuples

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters, and are immut able. Lists are sequences of elements of any data type, and are mutable. The third sequence type is the tuple. Tuples are lik e lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They're specified using parentheses instead of square brackets.
You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensu re that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be ch anged on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the return ed values is important, and a tuple ensures that the order isn't going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple return ed values from a function and store each value in its own variable.

From <https://www.coursera.org/learn/python-crash-course/supplement/A30RI/tuples>

```
>>> animals = ["Lion", "Zebra", "Dolphin", "Monkey"]
>>> chars = 0
>>> for animal in animals:
...     chars += len(animal)
...
>>> print("Total characters: {}, Average length: {}".format(chars, chars/len(animals))
```

```
winners = ["Ashley","Dylan","Reese"]
i=0
for winner in winners:
    print ("{} - {}".format(i+1,winner))
    i +=1
```

OR

```
winners = ["Ashley","Dylan","Reese"]
for index, winner in enumerate(winners):
    print ("{} - {}".format(index+1,winner))
```

The **enumerate** function **returns a tuple** for each element in the list. The **first** value in the tuple is the **index of the element** in the sequence. The **second** value in the tuple is the **element** in the sequence.

```
def full_emails(people):
    result = []
    for email, name in people:
        result.append("{} <{}>".format(name, email))
    return result
print(full_emails([('alex@example.com','Alex Diego'),('shay@example.com','Shay Brandt')]))
```

```
>>>
['Alex Diego <alex@example.com>', 'Shay Brandt <shay@example.com>']
>>>
```

```
def skip_elements(elements):
    new_list=[]
    for index, element in enumerate(elements):
        if index % 2 == 0:
            new_list.append(element)
    return new_list
print(skip_elements(["a", "b", "c", "d", "e", "f", "g"])) # Should be ['a', 'c', 'e', 'g']
print(skip_elements(['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach'])) # Should be ['Orange', 'Strawberry', 'Peach']
```

Iterating Over Lists Using Enumerate

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the elem ent to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first va lue of the tuple is the index and the second value is the element itself.

From <https://www.coursera.org/learn/python-crash-course/supplement/6E98c/iterating-over-lists-using-enumerate>

List comprehensions:
Let us create new lists based on sequences or ranges.

```
multiples = []
for x in range(1,10+1):
    multiples.append(x*7)
print(multiples)
```

OR

```
multiples = [x*7 for x in range(1,10+1)]
print(multiples)
```

So we can use this **technique** whenever we want to create a **list based on a range** like in this example. Or **based on the contents of** a list a tuple a string or **any other Python sequence**.

```
>>> languages = ["Python", "Perl", "Ruby", "Go", "Java", "C"]
>>> lengths = [len(language) for language in languages]
>>> print(lengths)
[6, 4, 4, 2, 4, 1]
>>> z = [x for x in range(0,101) if x % 3 == 0]
>>> print(z)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75
, 78, 81, 84, 87, 90, 93, 96, 99]
>>>
```

```
def odd_numbers(n):
    return [x for x in range(1,n+1) if x % 2 != 0]

print(odd_numbers(5))  # Should print [1, 3, 5]
print(odd_numbers(10)) # Should print [1, 3, 5, 7, 9]
print(odd_numbers(11)) # Should print [1, 3, 5, 7, 9, 11]
print(odd_numbers(1))  # Should print [1]
print(odd_numbers(-1)) # Should print []
```

Knowledge

Common sequence operations

Lists and tuples are both sequences and they share a number of sequence operations. The following common sequence operations are used by both lists and tuples:

- **len(sequence)** - Returns the length of the sequence.
- **for element in sequence** - Iterates over each element in the sequence.
- **if element in sequence** - Checks whether the element is part of the sequence.
- **sequence[x]** - Accesses the element at index [x] of the sequence, starting at zero
- **sequence[x:y]** - Accesses a slice starting at index [x], ending at index [y -1]. If [x] is omitted, the index will start at 0 by default. If [y] is omitted, the len(sequence) will set the ending index p osition by default.
- **for index, element in enumerate(sequence)** - Iterates over both the indices and the elements in the sequence at the same time.

List-specific operations and methods

One major difference between lists and tuples is that lists are mutable (changeable) and tuples are immutable (not changeable). There are a few operations and methods that are specific to changing data within lists:

- **list[index] = x** - Replaces the element at index [n] with x.
- **list.append(x)** - Appends x to the end of the list.
- **list.insert(index, x)** - Inserts x at index position [index].
- **list.pop(index)** - Returns the element at [index] and removes it from the list. If [index] position is not in the list, the last element in the list is returned and removed.
- **list.remove(x)** - Removes the first occurrence of x in the list.
- **list.sort()** - Sorts the items in the list.
- **list.reverse()** - Reverses the order of items of the list.
- **list.clear()** - Deletes all items in the list.
- **list.copy()** - Creates a copy of the list.
- **list.extend(other_list)** - Appends all the elements of other_list at the end of list

List comprehensions

A list comprehension is an efficient method for creating a new list from a sequence or a range in a single line of code. It i s a best practice to add descriptive comments about any list comprehensions used in your code, as their purpose can be diffic ult to interpret by other coders.

- **[expression for variable in sequence]** - Creates a new list based on the given sequence. Each element in the new list is the result of the given expression.
- Example: **my_list = [x*2 for x in range(1,11)]**
- **[expression for variable in sequence if condition]** - Creates a new list based on a specified sequence. Each element is the result of the given expression; elements are added only if the specified condition is true.
  - Example: **my_list = [x for x in range(1,101) if x % 10 == 0]**

From <https://www.coursera.org/learn/python-crash-course/supplement/sbRdF/study-guide-lists-operations-and-methods>

Quiz:

```
filenames = ["program.c", "stdio.hpp", "sample.hpp", "a.out", "math.hpp", "hpp.out"]

# Generate newfilenames as a list containing the new filenames
newfilenames=[filename.replace('.hpp','.h') for index,filename in enumerate(filenames)]

print(newfilenames) # Should be ["program.c", "stdio.h", "sample.h", "a.out", "math.h", "hpp.out"]
```

```
def group_list(group, users):
  members = ", ".join(users)
  return "{group}: {members}".format(group=group, members=members)

print(group_list("Marketing", ["Mike", "Karen", "Jake", "Tasha"])) # Should be "Marketing: Mike, Karen, Jake, Tasha"
print(group_list("Engineering", ["Kim", "Jay", "Tom"])) # Should be "Engineering: Kim, Jay, Tom"
print(group_list("Users", "")) # Should be "Users:"
```

fgWGKN

Dictionaries

Sunday, July 2, 2023 3:15 PM

```
>>> x = {}
>>> type(x)
<class 'dict'>
>>>
```

```
>>> file_counts = {"jpg":10, "txt":14, "csv":2, "py":23}
>>> print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 2, 'py': 23}
>>> file_counts["txt"]
14
>>> "jpg" in file_counts
True
>>> "html" in file_counts
False
>>> file_counts["cfg"] = 8
>>> print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 2, 'py': 23, 'cfg': 8}
>>> file_counts["csv"] = 17
>>> print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 17, 'py': 23, 'cfg': 8}
>>> del file_counts["cfg"]
>>> print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 17, 'py': 23}
>>>
```

Dictionaries Defined

Dictionaries are another data structure in Python. They're similar to a list in that they can be used to organize data into collections. However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer, float, or even tuples.

When creating a dictionary, you use curly brackets: **{}**. When storing values in a dictionary, the key is specified first, followed by the corresponding value, separated by a colon. For example, **animals = { "bears":10, "lions":1, "tigers":2 }** creates a dictionary with three key value pairs, stored in the variable animals. The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2. You can access the values by referencing the key, like this: **animals["bears"]**. This would return the integer 10, since that's the corresponding value for this key.

You can also check if a key is contained in a dictionary using the **in** keyword. Just like other uses of this keyword, it will return True if the key is found in the dictionary; otherwise it will return False.

Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists. You can add a new key value pair to a dictionary by assigning a value to the key, like this: **animals["zebras"] = 2**. This creates the new key in the animal dictionary called zebras, and stores the value 2. You can modify the value of an existing key by doing the same thing. So **animals["bears"] = 11** would change the value stored in the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the **del** keyword. By doing **del animals["lions"]** you would remove the key value pair from the animals dictionary.

From <https://www.coursera.org/learn/python-crash-course/supplement/2qpKG/dictionaries-defined>

```
>>> for extension in file_counts:
...     print(extension)
...
jpg
txt
csv
py
>>> for ext, amount in file_counts.items():
...     print("There are {} files with the .{} extension".format(amount, ext))
...
There are 10 files with the .jpg extension
There are 14 files with the .txt extension
There are 2 files with the .csv extension
There are 23 files with the .py extension
>>> file_counts.keys()
dict_keys(['jpg', 'txt', 'csv', 'py'])
>>> file_counts.values()
dict_values([10, 14, 2, 23])
>>> for value in file_counts.values():
...     print(value)
...
10
14
2
23
>>>
```

```
cool_beasts = {"octopuses":"tentacles", "dolphins":"fins", "rhinos":"horns"}
for animal, weapon in cool_beasts.items():
    print("{} have {}".format(animal, weapon))
>>>
Here is your output:
octopuses have tentacles
dolphins have fins
rhinos have horns
>>>
```

Because we know that each **key** can be present only **once**, dictionaries are a great tool for **counting elements** and **analyzing frequency**.

```
>>> def count_letters(text):
...     result = {}
...     for letter in text:
...         if letter not in result:
...             result[letter] = 0
...         result[letter] += 1
...     return result
...
>>> count_letters("aaaaa")
{'a': 5}
>>> count_letters("tenant")
{'t': 2, 'e': 1, 'n': 2, 'a': 1}
>>> count_letters("a long string with a lot of letters")
{'a': 2, ' ': 7, 'l': 3, 'o': 3, 'n': 2, 'g': 2, 's': 2, 't': 5, 'r': 2, 'i': 2, 'w': 1, 'h': 1, 'f
': 1, 'e': 2}
>>>
```

Iterating Over Dictionaries

You can iterate over dictionaries using a *for* loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary. If you want to access the corresponding values associated with the keys, you could use the keys as indexes. Or you can use the **items** method on the dictionary, like **dictionary.items()**. This method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

If you only wanted to access the keys in a dictionary, you could use the **keys()** method on the dictionary: **dictionary.keys()**. If you only wanted the values, you could use the **values()** method: **dictionary.values()**.

Object Oriented Programming

Sunday, July 2, 2023 11:37 PM

Object Oriented Programming

a way of thinking about and implementing our code.

a Flexible, powerful paradigm where a **class** (Apple) represents and defines **concept**, while an **object** (this apple) is an **instance of class**

The core **concept** of OOP comes down to **Attributes** and **Methods** *associated with a type.*

The **Attribute** are the **characteristics**(flavor and color) *associated to a type*.

The **Method** are the **functions**(bite, or slice - what you do with an object) *associated to a type*.

File has many attributes{Name, Size, Permissions, Date, ..etc}
There are actually so many different file attributes, that Python has multiple classes to deal with files.
The typical file object focuses on the file's contents, and so this object has a bunch of methods to read and modify what's inside the file.

Object-Oriented Programming Defined

In object-oriented programming, concepts are modeled as classes and objects. An idea is defined using a class, and an instance of this class is called an object. Almost everything in Python is an object, including strings, lists, dictionaries, and numbers. When we create a list in Python, we're creating an object which is an instance of the list class, which represents the concept of a list. Classes also have attributes and methods associated with them. Attributes are the characteristics of the class, while methods are functions that are part of the class.

From <https://www.coursera.org/learn/python-crash-course/supplement/kPaEL/object-oriented-programming-defined>

```
>>> type(0)
<class 'int'>
>>> type("")
<class 'str'>
>>>
```

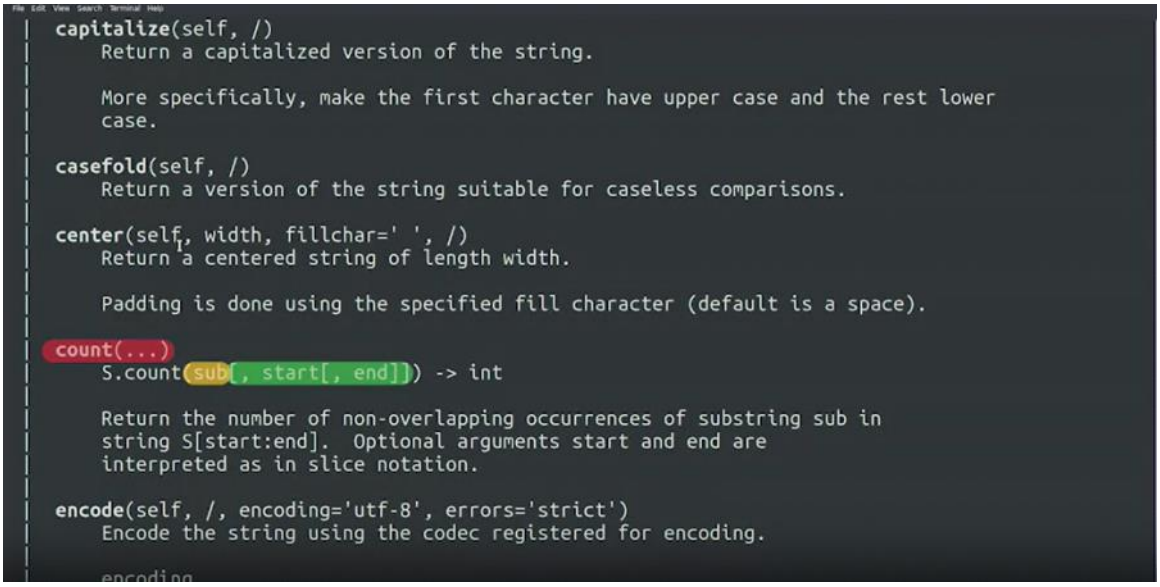
In this case, the only attribute is the content of the string.

```
>>> dir("")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__
', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '_
_new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeo
f__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswi
th', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdec
imal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isu
pper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase'
, 'title', 'translate', 'upper', 'zfill']
>>>
```

```
>>> help("")
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
```



Classes and Objects in Detail

We can use the **type()** function to figure out what class a variable or value belongs to. For example, **type(" ")** tells us that this is a string class. The only attribute in this case is the string value, but there are a bunch of methods associated with the class. We've seen the **upper()** method, which returns the string in all uppercase, as well as **isnumeric()** which returns a boolean telling us whether or not the string is a number. You can use the **dir()** function to print all the attributes and methods of an object. Each string is an instance of the string class, having the same methods of the parent class. Since the content of the string is different, the methods will return different values. You can also use the **help()** function on an object, which will return the documentation for the corresponding class. This will show all the methods for the class, along with parameters the methods receive, types of return values, and a description of the methods.

From <https://www.coursera.org/learn/python-crash-course/supplement/2YjP2/classes-and-objects-in-detail>

```
>>> class Apple:
...   pass
...
>>> class Apple:
...   color = ""
...   flavor = ""
...
>>> jonagold = Apple()
>>> jonagold.color = "red"
>>> jonagold.flavor = "sweet"
>>> print(jonagold.color)
red
>>> print(jonagold.flavor)
sweet
>>>
```

Dot notation

Lets you access any of the abilities the object might have (called methods) or information it might store (called attributes)

The attributes and methods of some objects can be other objects and can have attributes and methods of their own.
For example, we could use the upper method to turn the string of the color attribute to uppercase.

```
>>> print(jonagold.color.upper())
RED
>>> golden = Apple()
>>> golden.color = "Yellow"
>>> golden.flavor = "Soft"
>>>
```

```
class Flower:
  color = 'unknown'
rose = Flower()
rose.color = "red"
violet = Flower()
violet.color = "blue"
this_pun_is_for_you = "This pun is for you"

print("Roses are {},".format(rose.color))
print("violets are {},".format(violet.color))
print(this_pun_is_for_you)
```

Here is your output:

Roses are red,
violets are blue,
This pun is for you
Awesome! Very nice poem, if not a little cliche!

Classes and Methods

Monday, July 3, 2023 3:50 PM

```
>>> class Piglet:
...     def speak(self):
...         print("oink oink")
...
>>> hamlet = Piglet()
>>> hamlet.speak()
oink oink
>>>
```

```
>>> class Piglet:
...     name = "piglet"
...     def speak(self):
...         print("Oink! I'm {}! Oink!".format(self.name))
...
>>> hamlet = Piglet()
>>> hamlet.name = "Hamlet"
>>> hamlet.speak()
Oink! I'm Hamlet! Oink!
>>> petunia = Piglet()
>>> petunia.name = "Petunia"
>>> petunia.speak()
Oink! I'm Petunia! Oink!
>>>
```

Variables that have different values for different instances of the same class are called **instance variables.**

instance variable = attribute of the object (which is the name varaiable in this example)

```
>>> class Piglet:
...     years = 0
...     def pig_years(self):
...         return self.years * 18
...
>>> piggy = Piglet()
>>> print(piggy.pig_years())
0
>>> piggy.years = 2
>>> print(piggy.pig_years())
36
>>>
```

```
class Dog:
  years = 0
  def dog_years(self):
    return self.years * 7

fido=Dog()
fido.years=3
print(fido.dog_years())
```

Here is your output:
21
Awesome! You've now learned about writing your own methods!

What Is a Method?

Calling methods on objects executes functions that operate on attributes of a specific instance of the class. This means that calling a method on a list, for example, only modifies that instance of a list, and not all lists globally. We can define methods within a class by creating functions inside the class definition. These instance methods can take a parameter called **self** which represents the instance the method is being executed on. This will allow you to access attributes of the instance using dot notation, like **self.name**, which will access the name attribute of that specific instance of the class object. When you have variables that contain different values for different instances, these are called instance variables.

From <https://www.coursera.org/learn/python-crash-course/supplement/r3pJ0/what-is-a-method>

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold.color)
red
>>>
```

```
class Person:
    def __init__(self, name):
        self.name = name
    def greeting(self):
        # Should return "hi, my name is " followed by the name of the Person.
        return "hi, my name is {}".format(self.name)
# Create a new instance with a name of your choice
some_person = Person("Abdo")
# Call the greeting method
print(some_person.greeting())
```

Here is your output:
hi, my name is Abdo
Right on! You have successfully learned to assign attributes
to instances of class objects!

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold.color)
red
>>> print(jonagold)
<__main__.Apple object at 0x7fc4a58d5c18>
>>>
```

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold)
This apple is red and its flavor is sweet
>>>
```

Special Methods

Instead of creating classes with empty or default values, we can set these values when we create the instance. This ensures that we don't miss an important value and avoids a lot of unnecessary lines of code. To do this, we use a special method called a **constructor**. Below is an example of an Apple class with a constructor method defined.

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
```

When you call the name of a class, the constructor of that class is called. This constructor method is always named __**init**__. You might remember that special methods start and end with two underscore characters. In our example above, the constructor method takes the self variable, which represents the instance, as well as color and flavor parameters. These parameters are then used by the constructor method to set the values for the current instance. So we can now create a new instance of the Apple class and set the color and flavor values all in go:

```
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold.color)
Red
```

In addition to the __**init**__ constructor special method, there is also the __**str**__ special method. This method allows us to define how an instance of an object will be printed when it's passed to the print() function. If an object doesn't have this special method defined, it will wind up using the default representation, which will print the position of the object in memory. Not super useful. Here is our Apple class, with the __**str**__ method added:

```
>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
```

Now, when we pass an Apple object to the print function, we get a nice formatted string:

```
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold)
```

```
This apple is red and its flavor is sweet
```

This apple is red and its flavor is sweet
It's good practice to think about how your class might be used and to define a __str__ method when creating objects that you may want to print later.

From <https://www.coursera.org/learn/python-crash-course/supplement/z2XNm/special-methods>

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
>>> help(Apple)
```

```
Help on class Apple in module __main__:

class Apple(builtins.object)
 |  Apple(color, flavor)
 |
 |  Methods defined here:
 |
 |  __init__(self, color, flavor)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
(END)
```

Docstring

A brief text that explains what something does

```
>>> def to_seconds(hours, minutes, seconds):
...     """Returns the amount of seconds in the given hours, minutes, and seconds."""
...     return hours*3600+minutes*60+seconds
...
>>> help(to_seconds)
```

```
Help on function to_seconds in module __main__:

to_seconds(hours, minutes, seconds)
    Returns the amount of seconds in the given hours, minutes, and seconds.
(END)
```

```
class Person:
  def __init__(self, name):
    self.name = name
  def greeting(self):
    """Outputs a message with the name of the person"""
    print("Hello! My name is {name}.".format(name=self.name))
help(Person)
```

Here is your output:
Help on class Person in module submission:
class Person(builtins.object)
 | Methods defined here:
 |
 | __init__(self, name)
 | Initialize self. See help(type(self)) for accurate signature.
 |
 | greeting(self)
 | Outputs a message with the name of the person
 |
 | --
 | Data descriptors defined here:

```
 |
 | __dict__
 |     dictionary for instance variables (if defined)
 |
 | __weakref__
 |     list of weak references to the object (if defined)
```

Excellent! You've mastered the art of providing info using docstrings!

Documenting with Docstrings

The Python **help** function can be super helpful for easily pulling up documentation for classes and methods. We can call the **help** function on one of our classes, which will return some basic info about the methods defined in our class:

```
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
>>> help(Apple)
Help on class Apple in module __main__:
class Apple(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, color, flavor)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We can add documentation to our own classes, methods, and functions using **docstrings**. A docstring is a short text explanation of what something does. You can add a docstring to a method, function, or class by first defining it, then adding a description inside triple quotes. Let's take the example of this function:

```
>>> def to_seconds(hours, minutes, seconds):
...     """Returns the amount of seconds in the given hours, minutes and seconds."""
...     return hours*3600+minutes*60+seconds
...
```

We have our function called *to_seconds* on the first line, followed by the docstring which is indented to the right and wrapped in triple quotes. Last up is the function body. Now, when we call the help function on our to_seconds function, we get a handy description of what the function does:

```
>>> help(to_seconds)
Help on function to_seconds in module __main__:

to_seconds(hours, minutes, seconds)
    Returns the amount of seconds in the given hours, minutes and seconds.
```

Docstrings are super useful for documenting our custom classes, methods, and functions, but also when working with new libraries or functions. You'll be extremely grateful for docstrings when you have to work with code that someone else wrote!

From <https://www.coursera.org/learn/python-crash-course/supplement/LflUC/documenting-with-docstrings>

Classes and Methods Cheat Sheet (Optional)

Classes and Methods Cheat Sheet

In the past few videos, we've seen how to define classes and methods in Python. Here, you'll find a run-down of everything we've covered, so you can refer to it whenever you need a refresher.

Defining classes and methods

```
class ClassName:
    def method_name(self, other_parameters):
        body_of_method
```

Classes and Instances

- Classes define the behavior of all instances of a specific class.
- Each variable of a specific class is an instance or object.
- Objects can have attributes, which store information about the object.
- You can make objects do work by calling their methods.
- The first parameter of the methods (self) represents the current instance.
- Methods are just like functions, but they can only be used through a class.

Special methods

- Special methods start and end with __.
- Special methods have specific names, like __init__ for the constructor or __str__ for the conversion to string.

Documenting classes, methods and functions

- You can add documentation to classes, methods, and functions by using docstrings right after the definition. Like this:

```
class ClassName:
    """Documentation for the class."""
    def method_name(self, other_parameters):
        """Documentation for the method."""
        body_of_method
```

```
def function_name(parameters):
    """Documentation for the function."""
    body_of_function
```

Help with Jupyter Notebooks (Optional)

Help with Jupyter Notebooks

We've aimed to make our Jupyter notebooks easy to use. But, if you get stuck, you can find more information here.
If you still need help, the discussion forums are a great place to find it! Use the forums to ask questions and source answers from your fellow learners.
If you want to learn more about Jupyter Notebooks as a technology, check out these resources:

- Jupyter Notebook Tutorial, by datacamp.com
- How to use Jupyter Notebooks, by codeacademy.com
- Teaching and Learning with Jupyter, by university professors using Jupyter

From <https://www.coursera.org/learn/python-crash-course/supplement/5Ih9p/help-with-jupyter-notebooks-optional>