PALs CSC-152

Section 02, Professor Kim, M-W-F

Tony Stone

October 29, 2025

Lesson 0

1 Introduction

Welcome to Lesson 0! Rather than diving into a specific computer science topic, here are some background items to keep in mind. Many times throughout this document I will ask you to utilize your 'toolbox." Consider this lesson to be your hammer, and a set of nails.

2 Python Conventions

Like any programming language, Python has a plethora of conventions which are not true syntax. This means that avoiding these conventions will not have an impact on the performance of your code. So why follow them? Sticking to a set of conventions, particularly ones set for an entire language, make code generally more **readable**, and easy to work with (more on readability later).

Perhaps the most famous piece of programming convention is the naming convention; the way you will format your variables. In Python, the standard naming convention is **snake case**. This convention uses all lower case characters, separating them by underscores, like so:

```
# snake_case
```

Maybe it is a personal quirk, but I am of the mind that snake case is hideous. Throughout these lessons, I will stick to the **camel case** naming convention, like this:

```
# camelCase
```

Camel Case is written by having the first 'word' being lowercase, with an uppercase letter distinguishing each subsequent word. All of the words are packed together in a nice dense string. Though, when the first word begins with a capital letter, it is referred to as **pascal case**.

```
# camelCase
# --is not--
# PascalCase
```

Trust and believe that pedants will enjoy calling you out for 'mixing conventions' when you decide to give a class an uppercase letter, and claim your code follows camel case. Feel free to remind them: it literally does not matter.

3 Python Datatypes

Programming is a practice of **Data** manipulation. I do not often see an explicit definition of data in many resources—perhaps its obvious, but it was embarrassingly ambiguous to me—but data means literally anything that has any sort of

value. I could list examples, but we would be here all day. Data is more or less synonymous with 'stuff.'

However we like to classify this 'stuff' or data into different kinds of data. By doing so we are able to establish different features and qualities to form a set of unique **data types**. I will not give an exhausted review of each data type, but I will happily bullet the most relevant data types that are build into Python, and when necessary we'll look deeper.

First are the **atomic data types**, which are data types that represent the smallest division of data.

- **int**: This is an integer, or any whole number.
- **float**: This is a floating point number, or a number with a decimal point.
- **bool**: A boolean is a data type valued as either true or false, left or right, high or low, cats or dogs...

collection data types are data types are composed of atomic data types. As the name implies, they are various ways one can group atomic datatypes. Perhaps the most prominent among them is the **list**, which we will explore later! There are many other build-in collection datatypes, but lets not get rutted in that just yet.

I will take a quick moment to discuss the **str** data type, or string. These are made of a sequence of characters enclosed by quote marks: "", or by apostrophes: ''. Strings are sort of atomic, in that the smallest piece of a string is merely the empty string "", as it cannot be broken up into characters like other programming languages. But strings may also be treated as collections, as we will see later. Strings also have the quality of being **immutable**, or unchangeable. When we are 'altering' strings later, we are actually just making a new string! (it just looks like we are changing the old one).

4 Python Operations

Python utilities the expected suite of arithmetic operations. When combined in an arithmetic sequence, they follow the order of good old fashioned parenthesis > exponents > multiplication/division > addition/subtraction or **PEMDAS**.

Operator	Symbol
Addition	$+$
Subtraction	$-$
Multiplication	$*$
Division	$/$
Exponentiation	$**$

Aside from the standard arithmetic operations, Python offers other kinds of operations to be applied to numbers. Both of these operations round an otherwise decimal answer, into the nearest integer.

Operator	Symbol
Integer Division	//
Remainder Division (Modulo)	%

As you may have guessed, the modulo and integer division operations always have an answer of type int.

Operations are not limited strictly to numbers! There are many ways to apply operators to manipulate the data we begin with. I'll show a few, but try experimenting!

- String concatenation:

```
astring = "hi" + ", " + "how are you?" # Answer is "hi, how are you?"
```

- String multiplication:

```
bstring = "Stone" * 3 # Answer is "StoneStoneStone"
```

5 Resource Availability

These lesson pans are by no means the full extent of the resources available! I strongly suggest you visit the Introduction to Python GitHub Repository: github.com/TonyStone23/Introduction-to-Python. This will provide you with all of the **source code** of the examples you find in the lesson! When you come across an example within these lessons, feel free to hop over to the respective Python file, copy it, and play around with it!

6 Key Terms

- **Readable**: The measure of codes ability to be perceived.
- **Data**: All that holds a value of some kind.
- **Data Type**: The categories of data that grant values certain qualities.
- **Atomic Data Types**: The most fundamental components of data.
- **Collection Data Types**: Data types composed of atomic datatypes.
- **PEMDAS**: Parenthesis > exponents > multiplication/division > addition/subtraction.
- **mutability**: The ability to be changed.
- **source Code**: The original written code used to develop something.

Lesson I

1 Input, Processing, and Output

Many tasks in computer science involve collecting some kind of data, manipulating that data, and producing something from it. Imagine a bank, the input might be your account information, and request. The processing is all of the action the bank must do in order to make your request happen, or perhaps explain why they cannot complete your request. The output is that the bank completes your request, whether that be cash from a withdraw, or a receipt for a deposit.

1.1 Input

Here, our data is going to come from a user. We are going to prompt them for some kind of input. When we receive the user's input, we would like to store it in a variable, for further use, i.e. processing. This is how we will get input from a user. The prompt string we stick inside of input() will be output to the console, where the user can then type their answer and hit enter. It is generally good practice to include a break-line character '\n', or a simple space in the string, as Python does not automatically add any sort of spacing. Also, in the following examples, I will use a '|' to indicate the parts where a user would be typing.

```
1 userName = input("What is your name? ")
```

Output:

```
1 What is your name? |
```

1.2 Processing

Now that we have input, and it is stored, we can use it! There are some considerations, however. One consideration is that the input is a string, even when input a number. Consider:

```
1 num = input("input a number: ")
2 print(type(num))
```

Output:

```
1 <class 'str'>
```

class 'str' gets printed, even with a number-only input. Sometimes this is useful, when we are dealing with strings... But when we would like to use the numbers as actual numbers, we need to **typecast** them:

```
num = input("input a number: ")
num = int(num) # Typecast string input into an integer
print(type(num))
```

Output:

```
<class 'int'>
```

This now prints: **class 'int'** and can be used as such. This typecasting works for most datatypes: int(), float(), str(), etc.

Making sure your datatypes are in order is only a small snippet of what processing entails. Think of a calculator, the processing there is ... the calculating. We'll consider an example with arithmetic later on.

1.3 Output

So now what? We would like to produce some kind of output with the data that we had just collected. In keeping with the earlier example, we ask a user for their name, why not greet them!

```
userName = input("What is your name?")
print("Hello, " userName) # Add output to greet the user
```

Output:

```
What is your name? Tony|
Hello,  Tony
```

Now, when the user inputs their data, we then process it in order to output a greeting message.

2 Moving forward

What happens if the user inputs something absurd? This will most certainly crash our program! We will explore **Error-handling** in the future, this is just something to keep in the back of our minds. This lesson is meant to build a basis for console input and output. In later lessons, we will start to acquire tools that will make our input/output much more interesting! Be on the lookout!

3 Key Terms

- **Console**:
- **Typecast**: The process of converting a variable from one datatype to another.
- **Error-handling**: The process of dealing with potential issues that make code prone to crashing.

- ____ **code block**: A section of statements related statements in a code file. "____" is a placeholder for the different kinds of code blocks which we will certainly explore later.

Lesson II

1 The List Data Type

A **list** is a **collection** data type. It is a collection of various elements that can be stored in a single variable. In Python, we specify a list using brackets: "[]" and separate elements with commas ",". We will see many examples below!

1.1 List Operations

Sometimes we would like to access the items that are in a list. We can do that with **indexing**. In Python, and practically all programming languages, indexing begins at 0. So, the index of the first element is 0, the second is 1, the third is 2, etc. Here are some examples of indexing:

```
alist = ["my", "name", "is", "Tony"]
print(alist[0])
print(alist[2])
print(alist[3])
print(alist[-1])
```

Output:

```
my
name
is
Tony
Tony
```

Now that we have a list, the question is how are we going to use the list? When we index a list, we get that element, which we can then store into a variable to use for further operations.

```
nums = [1, 2, 3, 4, 5]
sumTwo = nums[3] + nums[4] # 4 + 5
print(sumTwo)
```

Output:

```
9
```

In addition to indexing, Lists have an additional operation known as **slicing**. This is very similar to indexing, only rather than indexing a single element, slicing indexes a range of elements. This operation returns a sub-list where the elements correspond to the index range. This range is specified by a colon, where the first number is the start of the slice, and the second number is the end. It is important to note that the first number is **inclusive** and the second is **exclusive**.

```
alist = ["my", "name", "is", "Tony"]
print(alist[1:4])
```

Output:

```
["name", "is", "Tony"]
```

As you can see in this example, the indices of our list only goes up to 3, but because the second number of our range is exclusive, we use 4 to extract all elements from the list.

Just like with strings, we can concatenate lists! We use the "+" operator, in order to combine them. Lets look at some examples, using all the tricks with lists that we've learned so far:

```
nums = [1, 2, 3, 4, 5]
nums = nums + [6]
print(nums)
nums = nums + nums
print(nums)
newNums = nums[0:2]+nums[10:12]
print(newNums)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
[1, 2, 5, 6]
```

1.2 List comprehension

List comprehension is a **Pythonic** way of creating lists. Do not feel intimidated by it's appearance. When we explore control structures and list creation in general, this will seem like a piece of cake. For now, let's just admire one of the many things that makes Python so special:

```
alist = [i for i in range(5)]
print(alist)
```

Output:

```
[0, 1, 2, 3, 4]
```

2 Key Terms

- **Collection**: A grouping of elements, with restrictions on inclusion based off of the specific type of collection.
- **List**: A collection data type used to store multiple items in a single variable.

- **Indexing**: Retrieving an item from a collection data type based off of the desired item's position.
- **slicing**: Indexing a collection with an index range, rather than a single index.
- **Pythonic**: Implementations that follow the conventions/quirks found in Python

Lesson III

1 Boolean Logic

1.1 Boolean?

Boolean refers to a binary variable with the possible values: "True" or "False." Boolean values open the door to the world of Boolean Algebra (You will learn more about this in later courses). We don't need to worry about rigorous boolean algebra right now, but we can discuss the boolean operations available to us. Let $T =$ "True" and $F =$ "False":

- **AND** T when both inputs are T

$$T \text{ AND } F = F$$

$$T \text{ AND } T = F$$

$$F \text{ AND } F = F$$

- **OR** T when at least one input is T

$$T \text{ AND } F = T$$

$$T \text{ AND } T = T$$

$$F \text{ AND } F = F$$

- **NOT** the opposite value of the input

$$\text{NOT } F = T$$

$$\text{NOT } T = F$$

Boolean algebra however is a very fascinating topic. There is so much more than just these operations, with very interesting outcomes and uses. Though we won't worry about it in the meantime, please keep it in your mind and look forward to it!

1.2 The Boolean Data Type

Naturally, in computer programming, the **Boolean data type** is a data type that represents these very values. Much like the int data type holds a $0, 1, ..., 9$, the boolean variable represents the state of "true" or "false."

1.3 Python's bool

Python has the boolean data type build in. The keyword that Python uses is **bool**. bools, and pythons implemented boolean logic allow us to utilize the above boolean operations in our programming. Lets see the same operations shown above, but as Python implementations!

- **AND**

```
# 'AND' operator is 'and'
print(True and False) # Output: False
print(True and True) # Output: True
print(False and False) # Output: False
```

- **OR**

```
# 'OR' operator is 'or'
print(True or False) # Output: True
print(True or True) # Output: True
print(False or False) # Output: False
```

- **NOT**

```
# 'NOT' operator is 'not'
print(not False) # Output: True
print(not True) # Output: False
```

These examples provide a great foundation for operations using bools, but this hardly scratches the surface. Often, we would like to use bools as a way to deal with certain conditions. But more on that later! Lets build up some ideas about other ways bools can be used:

- **Inequalities**

```
# value comparison using <, >, <=, =>
anum = 5
bnum = 6
print(anum > bnum) # Output: False
print(anum < bnum) # Output: True
```

- **Typecasting**

```
# typecasting bool
print(bool(1)) # Output: True (Values larger than 1 are true)
astr = ""
bstr = "any string"
alist = []
blist = ["at least one item"]
print(bool(astr)) # Output: False
print(bool(bstr)) # Output: True
```

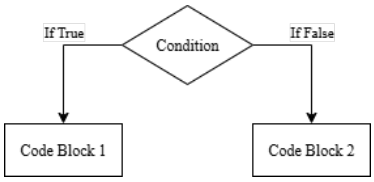


Figure 1: Branching

```
print(bool(alist)) # Output: False
print(bool(blist)) # Output: True
```

- **NOT**

```
# In case you need the opposite:
print(not astr) # Output: True
print(not bstr) # Output: False
print(not alist) # Output: True
print(not blist) # Output: False
# Note, you don't need to explicitly
# typecaste with 'bool' when using not
```

2 Branching

2.1 Decision structures

Say we would like to program something that will make a series of predefined decisions based on the input. We need a decision structure! Or a piece of code implemented in our program that will handle different things. We will stay rooted in python implementations. But do not worry, you will face the nuances of conditional statements later on in you CS journeys.

This is known as **branching**. Think of different possibilities of what gets run in the code as branches. See Figure 1. the word 'if' is very important, and is going to be crucial when it comes to implementing decisions in our code.

2.2 if, elif, and else

We use if when we want something to happen **if** something is True. Sometimes we would like this as a standalone check, maybe we want to change the state of a variable *if* something happens. We can do this with the keyword **if** and a piece of boolean logic:

```
age = int(input("How old are you"))
if age < 13:
    print(":D")
```

In this case, we prompt a user for their age. *If* they are 12 or younger, then we will print them a smiley face! But what about those 13 and up? We can greet them using the **else** keyword. What happens, is that if the **if statement** doesn't execute, the **else statement** will.

```
age = int(input("How old are you"))
if age < 13:
    print(":D")
else:
    print("hello")
```

Our original smiley face might be a little too kiddish for a 12 year old, we can add the **elif** keyword to extend our decision structure. elif is exactly what it sounds like; 'else-if' it is an else statement that uses an if statement!

```
age = int(input("How old are you"))
if age < 7:
    print(":D")
elif age < 13:
    print(":)")
else:
    print("hello")
```

3 Key Terms

- **Branching**: The extent of different outputs in a program.
- **Boolean**: A variable with possible values "True" or "False"
- **bool**: Python's boolean data type
- **AND, OR, NOT**: See above definitions
- **Decision Structure**: Code that provides different outputs based on inputs
- ___ **Statement**: By convention, a piece of code containing a keyword is described this way, e.g., "if Statement" or "else Statement"

Lesson IV

1 The Loop Structure

1.1 Motivation

Before we dive into the types of loop structures, we should first discuss why we may want to use a loop structure in the first place. Familiarize yourself with the term: **Increment**. Incrementing a given integer i is one of the most fundamental and crucial programming practices there is.

What incrementing mainly allows us to do is keep a count of something. This could be the length of a string, or maybe the number of attempts a user has at a game. Perhaps we would like to index a list in a specific element-wise fashion; incrementing an indexing variable is what we're looking for. This variable counts the positions of the list we've traversed, doesn't it?

We'll be focused on **while** and **for** loops. Perhaps in a bit of a backwards order, but we will: a. seamlessly transition from boolean logic into loop structures, and b. relate back to this section on traversing a list.

1.2 while

The while loop that makes a great use of boolean logic. With the **while** keyword, and then a piece of boolean logic, we can write a loop that starts as long as the condition we define is true, and end the loop when the condition is False. That may be a little abstract, so let's see an example:

```
    i = 0
    while i < 10:
        i += 1
    print(i)
```

Output:

```
10
```

This code will execute since $i = 0 < 10$, and when i finally increments such that $i = 10$ the statement $i < 10$ returns False, therefore breaking the while loop.

WARNING: We have all done it, but please try to avoid:

```
    i = 0
    while i < 10:
```

This is known as an **infinite loop**. Why? Allow me to pose you a question; will i ever be greater than 10? That's right, after your computer crashes.

Another Example:

```
    run = True
    i = 0
    while run:
        i += 1
        if i > 9:
            run = False
```

Output:

```
10
```

This is a more explicit means of the above code. Our 'boolean logic' is just the bool 'run' with the value "True" We increment i in the loop checking it's value. When i reaches 10, we change the value of 'run' from "True" to "False" which breaks the loop.

Lets explore a more complex example, a guessing game! We can use this while (*True statement*) to create a paradigm that allows a user any number of guesses. If you are not currently comfortable with function definition, and "import random" looks like wizardry, do not worry about it! One, we will cover functions very soon, so think of this as a soft introduction. Imports will continue to look like wizardry. The Python libraries are so rich and extensive that it *does* feel like magic. So I repeat, don't worry about it! It's just more for your toolbox!

```
import random
def guessingGame(allowedAttempts):
    numberToGuess = random.randint(1, 10)
    guesses = 1
    guessed = False
    while not guessed:
        guess = input(
            f"attempt {guesses}: What number am I thinking of? ")
        if int(guess) == numberToGuess:
            print("You guessed it!")
            guessed = True
        elif guesses == allowedAttempts:
            print("Better luck next time!")
            break # Exit the loop
        else:
            guesses += 1
```

We can exit a while loop using the **break** keyword. Sometimes we want to 'break' but only for the current iteration, this can be done with the **continue** keyword. We will see more examples of that later on!

Of course, a statement which is *not* false, is true. We can use this to our advantage. We can use this as the true-valued statement in our while loop definition. Most often we do this for **readability**. defining a loop with 'while not (*false statement*)' makes more intuitive sense than 'while (*true statement*' which is meant to represent a false one!

1.3 for

: One of the many things that is great about Python, is how it lends itself to natural language. As a programmer, you may think "I would like to do x for everything in this list." Well

```
blist = []
for item in alist:
    blist.append(item # With your desired operation x)
```

We see here that we have this variable item, which in each pass of this loop will take on the value of the 0th, 1st,..., nth element of alist. We'll do whatever operation x is, and then append it to our list b.

But do we need a whole new list? No!

```
for i in range(len(alist)):
    alist[i] = alist[i] # With your desired operation x
```

range is a keyword that returns a list of integers. It is called with range(a, b) and returns a range of $[a, b)$. If range(a, b) is just passed a number, it assumes a is zero. If we pass the length of our list along with it, we'll have a range of integers that covers the indices of our list, i.e., [0, len(alist)-1]. We use this index in order to index the element within the for loop,z to which we can do our operation, and then stick it right back in alist!

Some more concrete examples:

```
alist = [1, 2, 3]
blist = []
for item in alist:
    blist.append(item ** 2)
print(blist)
```

Output:

```
[1, 4, 9]
```

Or with indexing,

```
alist = [1, 2, 3]
for i in range(len(alist)):
    alist[i] = alist[i] ** 2
print(alist)
```

Output:

```
[1, 4, 9]
```

It is not inherently clear that there is a better or worse method. This will be entirely problem-dependent! Your job will be to apply the appropriate method for the given task.

We had mentioned before about something known as **list comprehension**. Then, it was relevant to the extent that it is interesting, and relevant to our use with lists. But now it is no mystery! We know what "for" is doing. That's right, list comprehension is an empty list with a for loop stuck inside of it. Lets build a list in two ways, to really see it in action.

```
alist = []
for i in range(1, 4): #Numbers 1-3
    alist.append(i**2)
print(alist)
```

Output:

```
[1, 4, 9]
```

but with list comprehension:

```
alist = [i**2 for i in range(1, 4)]
print(alist)
```

Output:

```
[1, 4, 9]
```

This is such a handy tool to have when you are seeking to create a list! Keeping a mental repository of such Pythonic statements is strongly recommended.

2 Key Terms

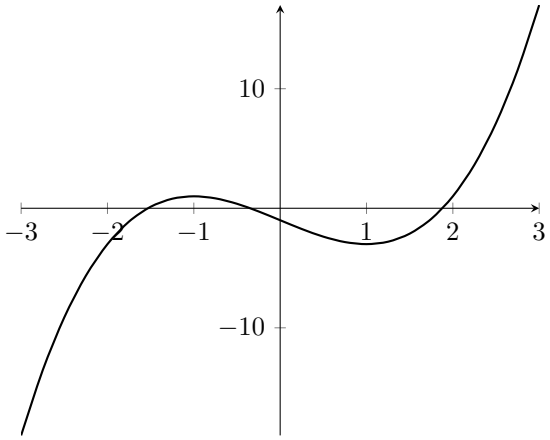
- **incrementing**: The act of repetitious addition of an integer.
- **while**: The keyword that begins a loop, and runs for as long as the associated statement is True.
- **break**: The keyword to exit a loop.
- **continue**: The keyword to end the current iteration, and begin at the next one.
- **Infinite loop**: A loop that is set to run for a condition which is unalterable from inside the loop.
- **Readability**: The ability to which human can parse code, and derive meaning.
- **for**: Loop structure that iterates based on a variables inclusion in a collection.
- **list Comprehension**(for loop perspective): A for loop that is nested within list brackets '[]' to generate a list.

Lesson V

1 Functions

1.1 Abstract Consideration of Functions

What exactly do we mean by "function?" Let's first think of this in the general sense. In math, when we are referring to a function, we are talking about a thing that takes an input, and gives an output. Consider a function f, where $f(x) = x^3 - 3x - 1$ Let's plot this!



Here, we see that for a given input x, there is a corresponding output $f(x)$. In the very abstract sense, we have a defined way of doing something ($f(x)$) to get something else (x).

1.2 Writing Functions

When we're looking to write our own functions, this paradigm is an important idea. We will usually have some *thing*, and we want a way to do something to it. I am being intentionally broad here. There is not much to limit what a function can, or cannot be.

To root ourselves back into computer science, suppose we have a string literal, and we want to search it for a specific character. This can be coded very plainly for the specific string.

```
1 char = 'a'
2 for c in "Hello":
3     print("found") if char is c else print("not found")
4     # Note the formatting of this if-else statement!
5     # Python sometimes has different ways to phrase stetements
6     # Another one for the toolbox!
```

But this only works for "hello" and the character "a," are we really going to re-write a for loop every time we want to check a character? Of course not! We have something: in this case a string and a character. We'd like to do something to it: search said string for said character. This is practically asking to be a function. Let's see that.

```
def findCharacterBasic(astring, achar):
    for c in astring:
        print("found") if achar is c else print("not found")
```

Basically the same thing right? But actually, this will allow us to input any string, and any character! The use of the function has allowed us to completely define this search operation in terms of variables. To so, we simply **call the function**:

```
findCharacterBasic("sue", "s"):
```

Output:

```
found
not found
not found
```

The variables that get passed into the function are known as a functions **Arguments**. The function we have written is known as a **void** function. A void function is a function that upon being called, that's it, we just call them! In our case, the function printed something in the console, but the function didn't give anything back to us. You can think of this as a one-way interaction, we call the function, and it goes off and does it.

Rarely do we want to print something out from within a function. Interactions with the console is the job of the main portion of our code. Console interactions are more-or-less hard coded, the beauty of functions is their flexibility, so don't bind them with console output.

1.3 Returning Functions

What we can do is write a function so that it will **return** a value (or many) so that we can store it, and do with it what we will. This is now a two-way interaction, where we call a function, and then it hands us something back.

A void function, or a function with a return statement is not inherently better or worse, and entirely use dependent. The world of **object oriented programming** (OOP) is made possible based on both of these function paradigms. We are not concerned with OOP right now, just keep this in the back of your mind!

Let's edit the above function to return something that we can store in a variable, and then make a decision about what to do with it.

```
def findCharacterEdited(astring, achar):
    found = False
    for c in astring:
        if achar is c:
            found = True
    return found

check = findCharacter("sue", "s")
if check is True:
    print("found")
else:
    print("not found")
```

Output:

```
found
```

Our for loop will run for every character in the string we provide it. We initialize a variable found to be false. As the loop iterates, if the character in the given iteration matches the character that we're checking for, it will assign it to be true, as we have found the character. Finally, our function returns the value of found, indicating the character's inclusion in the string.

When the return statement executes, this signifies the end of that function. This is particularly helpful with a conditional case; if something is true, return something, else, return something else! Both cases end the loop, but only return one thing. This will be much more helpful in later CS endeavors involving **recursion**. For now, just one return statement that returns values all properly organized within the function works just fine!

Let's backtrack a moment, a few times now I have mentioned functions returning several values. What does that look like?

```
def parseCharacter(astring, achar):
    found = False
    notChar = []
    for c in astring:
        if achar is c:
           found = True
        else:
            notChar.append(c)
    return found, notChar

checkIncluded, nots = parseCharacter("sue", "s")
print(checkIncluded, nots)
```

Output:

```
True ['u', 'e']
```

We are now keeping track of all of the characters that are not the character which we are parsing the string for. In the return statement, we can send this list back by separating variables with a comma. Notice how in the main portion where we are calling the function, we need to use two distinct variables in order to accept the functions output. We then print those two values, in no particularly fancy way.

2 Key Terms

- **Function**: A process that can be applied to types of items in a specified way.
- **Calling a function**: The act of stating the functions name to invoke it's action on an item.
- **Argument**: The variable that gets input into a function.
- **Void functions**: A function which does not return a value. 'void' or a word like 'void' is often used as a keyword in other languages.
- **return**: The keyword which allows functions to provide values.

Lesson VI

1 File Handling

1.1 Motivation

My personal experience with file handling has always felt so empty. It is traditional in introductory computer science courses to include sections on file handling; but why? It usually feels like a weird/niche veer to take in the middle of a course. Even worse, the explanation into the existence of the unit is always "so you can handle files" which a. obviously, and b. that is not a good reason! Why on Earth would we *want* to handle files?

Let's consider some applications of file handling! **Data Science** and file handling practically go hand in hand. Essentially all of the data you plan to use in a project will be interacted with through external files. If you are not so interested in data science—perhaps you are more into software development—the ability to process files is key to resolving user requests, and safety concerns. **Web applications** and **web development** are prominent use cases of file handling. Though admittedly, the choice of Python for such a task is debatable, so consider this as a broader conceptual endeavor into file handling, with practices that transcend any specific programming language.

1.2 The Flow of File Handling

The first step to file handling is opening the file. We'll course with a file called 'basicFile.txt':

```
My name is Tony Stone
```

As a general practice, we'll be opening the file, doing some sort of existence check, and closing the file. To open, we use the **open()** function, and store it in a variable. next, we'll use a print statement to check that it's worked, and then we will call the **close()** method to close our file.

```
file = open('basicFile.txt')
print(file)
file.close()
```

Output:

```
<_io.TextIOWrapper name='basicFile.txt' mode='r' encoding='cp1252'>
```

The argument for the open() function is a string literal, of the file path to the file. When the file is in the same folder as the one you are working in, the file is implied. But feel free to further organize your files, and be more explicit with file paths, separating each 'level' with a '/'. That will look something like:

```
'folderName/fileName.txt'
```

Now back to our example, what is all that junk at the bottom? what we've done is opened the file, and stuck it in a variable creating a file **Object**. Objects have defined methods that upon printing them, it displays something useful. hold on, what even is a method? A **method** is a function that is specific to an object. So, our file object has a method that makes printing it meaningful. As previously stated, we use another method, .close() in order to close the file. If you tried to call .close() on a plain string, or other type of object, you would get an error. since .close() is specific to file objects, it is called a method.

A good way to distinguish a method and a function is the preceding '.' for methods. This period is a pointer to the method, think of it like determining the job of a method by 'attaching' it to the object. Function definitions are not object-specific, and thus do not need specific pointers from objects.

Please do not get too bogged down in this, just observe the nuances, and be a little more mindful as you place these tools in your toolbox.

1.3 Reading Files

So now we can access, open, and close files, what are we going to do with it? Why don't we read the file. I'll create a folder called that contains our basic file, with a file that has some substance to it, so we can practice file paths along the way!

This is basically the same, but rather than printing the file object, lets use the **.read()** method to get all the stuff in the file, and then print that.

```
file = open('basicFolder/substantFile.txt')
fileContent = file.read()
print(fileContent)
file.close()
```

Output:

```
1 2 3 4 5
Howdy
:D
```

file.read() returns a string, so feel free to do all of your string manipulations to it! I quite enjoy that smiley face, lets just print that. fileContent is our string from file.read(), well go ahead and use the .split() method specifying were splitting at the '**n**' character. This returns a list which is our string divvied up at each line break. We'll index the last list element; our smiley face.

```
file = open('basicFolder/substantFile.txt')
fileContent = file.read()
print(fileContent.split('\n')[-1])
file.close()
```

Output:

```
:D
```

1.4 Writing to Files

Reading files is a very important tool that is integral to all kinds of tasks and workflows. But its only half the story! We want to have a way to document certain things about our code. Perhaps we are collecting data for a data science project, we need a place to store it all. Maybe we are running experiments and want a place to record results in a way more permanent than the console. File writing is the initiatory system for performing such tasks, and worth our exploring.

'afile.txt' will be our target, where we'll be applying some file writing techniques. Note that our file is located in the 'basicFolder' folder.

```
file = open('basicFolder/afile.txt', 'w')
file.write("Hello!")
file.close
```

Output:

```
Hello!
```

The 'w' argument in the function, open(), is one of several arguments that defines the way Python opens the function. 'w' specifically means we are opening the file in write mode. This will override any exist content, and even make a file if it doesn't exist! Go ahead, in the companion materials, feel free to delete 'afile.txt' and re-run the code. It's back!

We can also append to files. Have you ever written anything in one fell swoop? Of course not, any sort of record-keeping is a piecewise task, so we ought to have a way to do this. The 'a' argument allows us to simply append a text string to our document.

```
file = open('basicFolder/afile.txt', 'a')
file.write("Hello again!")
file.close
```

Output:

```
Hello!Hello again!
```

Yikes. let's be careful, adding a space at the start of our append string. We can overwrite all of this with the initial string with the 'w' argument, and then we can append our more precise string.

```
file = open('basicFolder/afile.txt', 'w')
file.write("Hello!")
file.close
```

```
4
5  file = open('basicFolder/afile.txt', 'a')
6  file.write(" Hello again!")
7  file.close
```

Output:

```
1  Hello! Hello again!
```

As we work on building our intuition and perception of code, this should feel somewhat off; and you'd be right! This is an extremely inefficient way to write files, we might as well just write them.

We can instead pass a bunch of strings as a list, so we don't have to keep opening and closing to keep in line with our intentions with the file. When doing this, well call the .writelines() method, and pass our list of strings as an argument. This may be slightly deceiving, as this method doesn't write 'lines' in the way we think of them, i.e., hitting the 'enter' key. It just mean's it will write with the multiple strings contained in the list. We'll use the handy-dandy '\n' to get some nice structure.

```
1  content = ["Hello! \n",
2             "Hello again! \n"
3             "Sh-boom"]
4  file = open('basicFolder/afile.txt', 'w')
5  file.writelines(content)
6  file.close
```

Output:

```
1  Hello!
2  Hello again!
3  Sh-boom.
```

We now have a guaranteed water-tight system for interacting with files!

...

2 Key Terms

- **Data Science**: The field of computer science related to data collection, processing, and analysis.
- **Web Development**: The field of computer science related to the developing, maintaining, and distributing of internet applications.
- **Method**: A function whose definition is specific to the object it is being used on.

Lesson VII

1 Motivation

No, not even close. While we have now comfortable rooted ourselves in the processes of reading and writing to files, you will come to appreciate the almost enigmatic world of programming, and the sheer immensity of all that can go wrong.

But have no fear! We have many tools at our disposal that will enable us to combat. These concepts are traditionally taught in tandem. Often file handling serves as a means to explore **exception handling**, or the process of cleanly dealing with the errors that arise when executing code. I find that this approach often leads to them being tied together in an almost synonymous relationship. But we already have motivation to work with files as we discussed in the last lesson! Our motivation to learn about file handling exceeds far past the application of file handling. Thus, the topics are provided in separate lessons. Some of the examples features here will involve files, but also other kinds of issues that may come up!

1.1 Breaking the Fourth Wall

If you will allow me a moment to speak to you directly and candidly; more often than not you will not be using some exception handling. The code that you plug into a python terminal to quickly test a code snippet will not require exception handling. A majority of assignments you will encounter in coursework, or future educational endeavors will not require exception handling. Exception handling is meant to be a way to make your code water-tight.

Suppose you wanted a bullet-proof vehicle, and you started by steel-plating a go-kart frame. How nonsensical! The go-kart must be built up into something like a vehicle first; something reliable with a few features. After that, we can begin thinking about steel plating, otherwise it is overkill. 90% of the programming you will do is riding on go-karts. Save Exception handling for when you plan to juice one up.

Also, also, I am referring to 'Exception' as the stuff that goes wrong in code. There is a difference between an Error, and Exception, a this and a that. But the Juice of diving into it is not worth the squeeze. When we say **Exception**, we are talking about the stuff that goes wrong.

2 Exception Handling

2.1 the Try Statement

The canonical math 'no-no' is division by zero. Let's do it!

```
1 print(3/0)
```

Output:

```
    print(3/0)
          ~^~
ZeroDivisionError: division by zero
```

Unsurprisingly, this is a no-can do. Now is a good time to introduce the concept of a **stack trace**. For whatever reason, a stack trace is not typically acknowledged until software engineering-type courses, but I find that extremely unhelpful. Your stack Trace is the tool that lets you parse whatever caused your program to crash. Please spend some time just simply reading it. Even if you do not pull a single thing from it, get familiar with the format, and try to follow what it is reporting.

Anyway, the simple division method simply will not do. It never will, but we don't need it too, we just need our program to keep chugging. The **try** statement is what will let us do that. The **try block** is the section of sketchy code we put after a try statement. Its almost like asking our program to perform a task, instead of forcing it.

However a try statement is not a standalone structure, it needs to have a corresponding keyword that helps the program stay on it's feet. We have some options, keywords **finally**, and **except**. Finally is used to mark a section of code that will execute regardless of the execution of the try block. This can be as easy as a pass statement, but we'll print a message for clarity.

```
print("\nExample Two:")
try:
    print(3/0)
finally:
    print("Program executed")
```

Output:

```
Program executed
...
    print(3/0)
          ~^~
ZeroDivisionError: division by zero
```

As you can see, our program executes, but it still delivers us all of the stack trace jargon. This is the purpose of the except keyword. By OOP convention, we 'catch' the error, and we can format it such that it is more useful to us. When an exception is **raised**, we'll take it and give it a name, and format it into a message.

```
try:
        print(3/0)
    except Exception as e:
        print(e,"happened")
```

```
    finally:
        print("Program executed")
```

Output:

```
division by zero happened
Program executed
```

Here, we call our exception 'e', and with our print statement, we are informed of our erroneous division in a much more readable way. Also, a standalone except statement is totally fine, the finally statement just allows you some more control.

2.2 The Dark Arts

I emphasized 'raised' but completely blew past it. In general, 'raise' is simply the term we use for an error that comes out of executing code. However, for any aspiring mage seeking to dabble in the dark arts, raise is in fact a keyword. You can practice tried-and-true arcana (Python Documentation) to raise an Error stock to Python, or you can take to the tomes and artifice new ones (write a class).

Let's look into raising our own exceptions. The ones stock to Python are almost certainly more than enough, we'll leave exception crafting to the thaumaturges.

We'll write a function, but we'll be picky, and raise an error if we deem the input unsuitable. Consider the case of a function that adds two numbers. If our user inputs a string (which remember, they will) we need to handle that.

```
def addNum(a, b):
    if type(a) is str or type(b) is str:
        raise ValueError("That is not a number!")
    return a + b

sum = None
try:
    sum = addNum(1, 2)
except ValueError:
    print("Ugh-oh")
finally:
    print(sum)
```

Output:

```
3
```

Now for the case in which our user inputs a string, regardless of the safeguards and warnings put in place beforehand.

```python
def addNum(a, b):
    if type(a) is str or type(b) is str:
        raise ValueError("That is not a number!")
    return a + b

sum = None
try:
    sum = addNum(1, "astring")
except Exception:
    print("Ugh-oh")
finally:
    print(sum)
```

Output:

```
Ugh-oh
None
```

Please put a more helpful message than "ugh-oh," but this clearly indicates that a problem has arisen. We then show that sum has not updated with the finally block. Note, the error we raise, and the try-except/finally logic is separate. We also raise a **ValueError** specifically, just make sure that the error you raise, and the error you handle is cohesive to some extent. We'll skip the definitions, but ValueError is a subclass of Exception, so we can simply except Exception, and ValueError will neatly fit into that.

3 Back to File Handling

Let's take what we've learned about exception in general handling and relate it back to the much more useful application of file handling. Let's (not) open the file 'nonExistantFile.txt.'

```python
file = open('basicFolder/nonExistantFile.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

Output:

```
    file = open('basicFolder/nonExistantFile.txt', 'r')
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory:
'basicFolder/nonExistantFile.txt'
```

Imagine trying to download a wealth of data for a project, and right in the middle of it this happens. Well we don't have to, because we have acquired the tools we need to navigate such an event.

```
fileName = 'basicFolder/nonExistantFile.txt'
contents = None
try:
    file = open(fileName, 'r')
    contents = file.read()
    file.close()
except FileNotFoundError:
    print(fileName, "doesn't exist")
finally:
    print(contents)
```

Output:

```
basicFolder/nonExistantFile.txt doesn't exist
None
'basicFolder/nonExistantFile.txt'
```

Just for our sanity, let's look back and make sure this could work, with our old 'substantFile.txt.'

```
fileName = 'basicFolder/substantFile.txt'
contents = None
try:
    file = open(fileName, 'r')
    contents = file.read()
    file.close()
except FileNotFoundError:
    print(fileName, "doesn't exist")
finally:
    print(contents)
```

Output:

```
1 2 3 4 5
Howdy
:D
```

4 Key Terms

- **Exception Handling**: The process of dealing with exceptions.
- **Exception**: A problem that is prone to crashing code.
- **Stack Trace**: Report of the errors in the code, specifically a traced path to the original error.
- **try**: The keyword to organize exception prone code.

- **except**: The keyword which actually handles the exception, and allows for an alternative course of action in the case of an exception.
- **finally**: The keyword which executes a block of code regardless of the outcome from the try-except block.
- **raise**: The word for the realization of an exception, (a keyword if done intentionally).

5 Availability

- In Lecture:
  - Wednesday: 10:00am - 11:00am
- PAL Sessions:
  - Monday: 2:00pm - 3:00pm
  - Wednesday: 9:00am - 10:00am
  - Wednesday: 2:00pm - 3:00pm

References

[1] Tony Gaddis. *Starting Out with Python*. Pearson, 5th edition, 2021.

[2] Bradley N. Miller and David L. Ranum. Problem solving with algorithms and data structures using python. https://runestone.academy/ns/books/published/pythonds/index.html, 2014. Interactive online edition — accessed: 2024.