ICS 32 Winter 2022 Project #1: *Digging in the Dirt*

Due date and time: *Wednesday, January 19, 11:59pm*

Introduction

While there are clear differences between the operating systems that run personal computers, one of the things that they all have in common is the notion of a *file system*, whose role is to manage the creation, arrangement, updating, and deletion of files. Storage devices such as hard drives and USB sticks are actually a lot more complicated than they look, but a file system provides an abstraction over that complexity, replacing it with a few simple concepts like *files*, *directories*, and *paths*.

Like many programming language libraries, Python's standard library provides a pre-built implementation of a variety of file system operations. It is possible to create files, copy and move files, search in directories, and manipulate paths. Once you know the abstract concepts that a file system is centered around, you can use Python's standard library to access a file system in terms of those same abstract concepts. We've seen already that there is a **str** type in Python that knows how to manage sequences of characters and a **float** type that knows how to store and manipulate floating-point numbers; because these are built into Python, you can use them without having to know every tiny detail of how they work internally. Similarly, the Python standard library offers types like **Path** that can handle the details of manipulating a file system, meaning that you won't have to understand every detail of how they work internally in order to make good use of them.

This project will give you a chance to explore a few parts of the Python standard library that you may not have seen before; provide practice in reading and digesting technical documentation in order to determine what functions in the standard library are appropriate in helping you to solve a problem; introduce you to some features of Python you might not have had the opportunity to use before, such as exception handling; and ask you to use recursion to traverse a recursive data structure (in this case, the file system).

File systems

Whether you run Windows, macOS, or some flavor of Linux, you've no doubt had experience with at least some of the following concepts, though you may not be familiar with all of the terminology, and not all of you will have seen the same subset of these concepts. To ensure that we're all on the same page, here are some things you'll need to know about file systems.

- A *file system* is software that manages how information is stored on a storage device such as a hard drive.
- Information is stored in *files*, which are containers in which a sequence of *bytes* are stored. Each byte is effectively a sequence of eight "digits" that are all either 1 or 0; each of these is called a *bit*. The bytes in each file are interpreted differently depending on what kind of file it is (i.e., what the program reading the file expects it to contain, such as text, an image, a song, or a video).
- Each file has a *filename* that can be used to identify it. While some operating systems treat files differently depending on their names — most often based on the *extension*, which is the part of the filename that

follows the last dot (e.g., **.doc** in the filename **invoice.doc**) — there is not necessarily a relationship between the name of a file and the kind of data it contains. Try renaming one of your word processing documents as **image.jpg** and opening it in your favorite photo editing software; the software may initially believe it's a JPEG file (i.e., an image) because of the name ending in **.jpg**, but it will quickly realize that it's not when the contents of the file are something entirely different, and will probably show you some kind of error message.
    - *Side note: If you're running Windows on your own machine, have you completed the Python installation instructions in Project #0 yet? In particular, Step 1 of the installation instructions for Windows is not to be overlooked; it's going to be **really important** in the context of this project! Windows, by default, hides filename extensions in its user interface, but we're going to need to see the filenames for what they are. If you haven't done this, DO IT NOW! You'll be glad you did — and not just in your work on this project.*
- Files are arranged into *directories* (also called *folders*, but we'll settle on one terminology and call them "directories," since that's the terminology used in Python's standard library). Like files, directories have names.
- Directories can contain not only files but other directories, which can, in turn, contain other directories, and so on. A directory inside of another directory is generally called a *subdirectory*.
- The rules that govern directories are, more or less, the same as the rules that govern the directories inside of them, except for one special directory called the *root*, which forms a kind of starting point for a search; the root directory is the one directory on a device that is not inside another directory.
- More than one file or directory on a device can have the same name, provided that they are stored in different directories; they are instead identified more precisely by a *path*, which indicates the sequence of directories, starting from the root, one would have to descend into in order to find a file or directory. While the concept of a path transcends operating systems, they are written slightly differently on different ones.
- Instead of storing a file in a directory, you can also store a *symbolic link* that points to another file stored in some other directory, which allows the same file to appear in more than one directory. Similarly, a directory can contain a symbolic link to another directory (and this can get a little bit strange, such as directories that link back to their parent or even to themselves, but this kind of weird setup isn't recommended).
    - This project won't require you to handle symbolic links, though you're welcome to take a crack at it if you're so inclined. Do be aware, though, that it may not be easily possible to create them on Windows — it can be done, but requires tweaking security settings on some versions of Windows, and would require administrative access to your machine, which means you wouldn't be able to do it on machines for which you don't have that access (e.g., the machines in the ICS labs).

If most of these terms seem unfamiliar, it's worth doing a bit of research and experimentation to be sure you understand what they are before you proceed too far through this project. In addition to being useful in the context of the problem at hand, this is good practical knowledge that you'll need if you want to be successful as a programmer.

The program

The program you'll be writing for this project is one that can find and display the paths of all of the files in a directory (and, potentially, all of its subdirectories, and their subdirectories, and so on), and then take action on some of those files that have interesting characteristics. Both the notion of *interesting characters* and *taking action* will be configurable and each can work in a few different ways, but the core act of finding files will always be the same. One of your goals should be to avoid rewriting the same code over and over again (e.g., multiple functions that each perform a search for files with slightly different characteristics) whenever possible; in ICS 32, we begin to concern ourselves more strictly with design issues, keeping an eye on how *best* to solve a program, as opposed to writing something that "just works."

The input

Your program will take input from the console in the following format. It should not prompt the user in any way. The intent here is not to write a user-friendly user interface; what you're actually doing is building a program that we can test *automatically*, so it's vital that your program reads inputs and writes outputs precisely as specified below.

- First, the program reads a line of input that specifies which files are eligible to be found. That can be specified in one of two ways:
  - The letter **D**, followed by a space, followed (on the rest of the line) by the path to a directory. In this case, all of the files in that directory will be under consideration, but no subdirectories (and no files in those subdirectories) will be. (You can think of the letter **D** here as standing for "directory.")
  - The letter **R**, followed by a space, followed (on the rest of the line) by the path to a directory. In this case, all of the files in that directory will be under consideration, along with all of the files in its subdirectories, all of the files in their subdirectories, and so on. (You can think of the letter **R** here as standing for "recursive.")
  - If this line of input does not follow this format, or if the directory specified does not exist, print the word **ERROR** on a line by itself and repeat reading this line of input; continue until the input is valid.
- Next, the program prints the paths to every file that is under consideration. Each path is printed on its own line, with no whitespace preceding or following it, and with every line ending in a newline. Note, also, that the order in which the files' paths are printed is relevant; you must print them in the following order:
  - First, the paths to all of the files in the directory are printed. These are printed in *lexicographical order* of the file's names. (More on lexicographical order a bit later, but note that this is the default way that strings are sorted.)
  - Next, if the files in subdirectories are being considered, the files in each of the subdirectories are printed according to the same ordering rules here, with all of the files in one subdirectory printed before any of the others, and with the subdirectories printed in lexicographical order of their names.
- Now that the program has displayed the paths of every file under consideration, it's time to narrow our search. The program now reads a line of input that describes the *search characteristics* that will be used to decide whether files are "interesting" and should have action taken on them. There are five different characteristics, and this line of input chooses one of them.
  - If this line of input is the letter **A** alone on a line, all of the files found in the previous step are considered interesting.
  - If this line of input begins with the letter **N**, the search will be for files whose names exactly match a particular name. The **N** will be followed by a space; after the space, the rest of the line will indicate the name of the files to be searched for.
    - Note that filenames include extensions, so a search for **boo** would not find a file named **boo.doc**.
  - If this line of input begins with the letter **E**, the search will be for files whose names have a particular *extension*. The **E** will be followed by a space; after the space, the rest of the line will indicate the desired extension.
    - For example, if the desired extension is **py**, all files whose names end in **.py** will be considered interesting. The desired extension may be specified with or without a dot preceding it (e.g., **E .py** or **E py** would mean the same thing in the input), and your search should behave the same either way.
    - Note, also, that there is a difference between what you might call a *name ending* and an *extension*. In our program, if the search is looking for files with the extension **oc**, a file named **iliveinthe.oc** would be found, but a file named **invoice.doc** would not.
  - If this line of input begins with the letter **T**, the search will be for text files that contain the given text. The **T** will be followed by a space; after the space, the rest of the line will indicate the text that the file should contain in order to be considered interesting.

        - For example, if this line of input reads **T while True**, any text file containing the text "while True" would be considered interesting.
        - One thing to note is that not all files are text files, but that you can't determine that by their name or their extension. Any file that can be opened and read as a text file is considered a text file for our purposes here, regardless of its name. Any file that cannot be opened and read as a text file should be skipped (i.e., it is not considered interesting).
    - If this line of input begins with the character **<**, the search will be for files whose size, measured in bytes, is less than a specified threshold. The **<** will be followed by a space; after the space, the rest of the line will be a non-negative integer value specifying the size threshold.
        - For example, the input **< 65536** means that files whose sizes are no more than 65,535 bytes (i.e., less than 65,536 bytes) will be considered interesting.
    - If this line of input begins with the character **>**, the search will be for files whose size, measured in bytes, is greater than a specified threshold. The **>** will be followed by a space; after the space, the rest of the line will be a non-negative integer value specifying the size threshold.
        - For example, the input **> 2097151** means that files whose sizes are at least 2,097,152 bytes (i.e., greater than 2,097,151 bytes) will be considered interesting.
    - If this line of input does not match one of the formats described above, print the word **ERROR** on a line by itself and repeat reading a line of input; continue until the input is valid. Note that it is not an error to specify a search characteristic that matches no files; it's only an error if this line of input is structurally invalid (i.e., it does not match one of the formats above).
- Next, the program prints the paths to every file that is considered interesting, based on the search characteristic. Each path is printed on its own line, with no whitespace preceding or following it, and with every line ending in a newline. The paths should be printed using the same ordering rules as the last time you printed them (i.e., lexicographical ordering, as described above), though, of course, you will likely print fewer this time, since not every file will necessarily meet the search characteristic.
- If there were no interesting files, the program ends; there is no action to take.
- Now that we've narrowed down our search, it's time to take action on the files we found. The actions are to be taken on the files in the same order as you printed them previously. The program now reads a line of input that describes the *action* that will be taken on each interesting file. There are three different actions, and this line of input chooses one of them.
    - If this line of input contains the letter **F** by itself, print the first line of text from the file if it's a text file; print **NOT TEXT** if it is not.
    - If this line of input contains the letter **D** by itself, make a duplicate copy of the file and store it in the same directory where the original resides, but the copy should have **.dup** (short for "duplicate") appended to its filename. For example, if the interesting file is **C:\pictures\boo.jpg**, you would copy it to **C:\pictures\boo.jpg.dup**.
    - If this line of input contains the letter **T** by itself, "touch" the file, which means to modify its last modified timestamp to be the current date/time.
    - If this line of input does not match one of the formats described above, print the word **ERROR** on a line by itself and repeat reading a line of input; continue until the input is valid.
- Once an action has been taken on all files, the program ends.

A few additional notes

Since one of the goals of this project is to introduce you to the use of recursion to solve real problems, the search that includes subdirectories and their subdirectories must be implemented as a recursive Python function that processes all of the files in a directory and then processes any subdirectories recursively. (Note that this rules out certain features of the Python standard library — searches like this are actually built into the library, but would circumvent the learning goal here. More on that later.)

Outside of the occurrence of symbolic links, which we're ignoring for the purposes of this project, directory structures are hierarchical (i.e., directories have subdirectories inside of them, and those subdirectories have the same basic structure as their "parents").

An example of the program's execution

The following is an example of the program's execution, as it should work when you're done. Boldfaced, italicized text indicates input, while normal text indicates output. The directories and files shown are hypothetical, but the structure of the input and output is demonstrated as described above.

To reiterate a point from earlier, your program should not prompt the user in any way; it should read input, assuming that the user is aware of the proper format to use.

```
R C:\Test\Project1\Example
C:\Test\Project1\Example\test1.txt
C:\Test\Project1\Example\test2.txt
C:\Test\Project1\Example\Sub\meee.txt
C:\Test\Project1\Example\Sub\test1.txt
C:\Test\Project1\Example\Sub\youu.txt
C:\Test\Project1\Example\Zzz\zzz.py
N
ERROR
N test1.txt
C:\Test\Project1\Example\test1.txt
C:\Test\Project1\Example\Sub\test1.txt
Q
ERROR
F
This is a line of text
Hello, my name is Boo
```

Portability

It should be possible to run your program on any operating system — Windows, macOS, Linux, etc. — so long as there is a Python 3.9 implementation for it. By and large, this doesn't require much special handling: Python is already reasonably independent of its platform. However, since you're dealing with a filesystem, you do have to be cognizant of how filesystems are different from one operating system to another. Most notably, paths are written differently:

- On Windows, paths tend to be a *drive letter*, followed by a colon, followed by a sequence of directory names separated by backslashes, such as **D:\Docs\UCI\ICS32\Homework**.
- On macOS, Linux, and other flavors of Unix, paths tend to be a sequence of directory names preceded by forward slashes instead, such as **/home/student/uci/ics32/homework**.

The way to isolate this distinction is to find tools in Python's standard library that isolate them for you. Don't store paths as strings; store them as path objects instead. (More on this below.)

Handling failure

You'll want to handle failure carefully in this program. In general, your program should not crash just because one activity fails; it should instead quietly skip the offending file or directory and continue, if possible. For example:

- If, during the search, accessing some directory fails, the search should still continue attempting to access other directories.
- If taking action on an interesting file fails, the program should continue processing additional interesting files. The only exception to this rule is when printing the first line of text from a file, in which case you would print **NOT TEXT** if the file is not a text file.

Organization of your program

There are a few things to note about how your program is organized.

- Your program must be written entirely in a single Python module, in a file named **project1.py**. Note that capitalization and spacing (i.e., no letters in the filename are capitalized and there are no spaces) are important here; they're part of the requirement.
- Executing your **project1** module — by, for example, pressing F5 or selecting **Run Module** from the **Run** menu in IDLE — must cause your program to read its input and then print its output. It must not be necessary to call a function manually to make your program run.
- Importing your module manually using an **import** statement should not cause your program to run. The way to differentiate between these scenarios is to write an **if** statement, outside of any function (and usually at the bottom of your module), that looks like this:

  ```
  if __name__ == '__main__':
      the_things_you_want_to_happen_only_when_the_module_is_run
  ```

  The expression **__name__ == '__main__'** will only return **True** within a module that was originally executed using F5 in IDLE; it will return **False** in any other module (e.g., in a module that has been imported instead).

A word about documentation

As you likely did in ICS 31, you are required to include type annotations and docstrings on every one of your functions. This will not only make your design clear to us when we're grading your project, but will also clarify your own design for yourself; details like this will help you to read and understand your own program. I find, as a general rule, that I will always write (or, at the very least, consider and understand) what the types my parameters and return value will be *before* I write any function; knowing the types is part of knowing what the function is supposed to do.

For example, if you were to write a function **find_max** that finds the maximum integer in a list of integers, you might start the function this way:

```
def find_max(numbers: list[int]) -> int:
    '''Finds and returns the maximum number in a list of numbers'''
    ...
```

Other aspects of how this project (and others) will be graded are described on the front page of the Project Guide. Keep in mind, in particular, that part of your score will be based on how well your program is designed; the front page of the Project Guide describes this in more detail.

An important design goal

As you write your program, you'll want to be on the lookout for complex functions that can be made simpler by breaking them up into multiple smaller functions. Taking complexity and putting it into a function with a well-chosen name and clearly-named parameters is a great way to tame that complexity; this is the first step in building an ability to write significantly larger programs than you've written so far, so be sure that you're getting some practice in taking that step in this project. One of the criteria we'll use in grading your project is to assess the quality of its design; in this project, the largest factor affecting quality is the extent to which functions were broken up when they are long or cumbersome.

A good rule of thumb is to consider what you would say if I asked you "What does this function do?" If the answer is more than a sentence or so — and probably a short one, at that! — it's probably doing too much, and might better be implemented as multiple functions that each do part of the job. (One of the good things about writing docstrings in your functions is that it forces you to test this; if your docstring needs to be especially long, your function is probably too complex.)

Wait... what kind of crazy user interface is this?

Unlike programs you may have written in the past, this program has no graphical user interface, or even an attempt at a "user-friendly" console interface. There are no prompts asking for particular input; we just assume that the program's user knows precisely what to do. It's a fair question to wonder why there isn't any kind of user interface. Not all programs require the user-friendly interfaces that are important in application software like Microsoft Word or iTunes, for the simple reason that humans aren't the primary users of all software. As we'll see going forward in this course, much of what happens in sofware is programs talking directly to other programs; in cases like that, each program generally presumes that the other one is aware of the right way to communicate, and will give up quickly if the other program isn't ready to play by those rules.

Now consider again the requirements of the program you're being asked to write for this project. It waits for requests to be sent in via the console — though they could almost as easily be sent across the Internet, if we preferred, which we'll see in the next project — in a predefined format, then responds using another predefined format. Our program, in essence, can be thought of in the same way as a web server; it's the engine on top of which lots of other interesting software could be built. When an attempt in being made to search for a file, that could be coming from a web form filled out by a user, from another program that needs to perform a search, or from a human user in the Python shell.

While we won't be building these other interesting parts, suffice it to say that there's a place for software with no meaningful user interface; it can serve as the foundation upon which other software can be built. You can think of your program as that foundation.

Additionally, we're using a strategy like this one to assist us in automating the grading of the correctness of your project. Since everyone's input and output have to be formatted in the same way, we will be able to grade your output without manually looking at every line.

The Python Standard Library documentation

At **python.org**, you'll find a comprehensive set of documentation about the Python language and its standard library. Two good places to start are these pages:

- Python 3.10 documentation
- Python 3.10 Standard Library documentation

You'll probably need to spend a fair amount of time, especially early in your work on this project, looking at the Standard Library documentation and assessing what functions in that library might be able to assist you in your solution. The Standard Library documentation is mostly broken down by module. It's tough sometimes to know where to look, as there are so many modules, so we'll sometimes try to point you in the right direction. For this project, you'll want to pay special attention to (at least) these modules: **pathlib**, **os**, and **shutil**, though you might also find useful tools in other modules. Feel free to poke around and see what's available; if it's in the Standard Library, you can feel free to use it, except for one limitation pointed out below.

Note, too, that part of understanding what's available in the Standard Library is careful, targeted experimentation. If you're not sure what a library function does just by reading its documentation, you can fire up IDLE and experiment with it; one of the nice things about an interpreted programming language like Python is that this experimentation can be done in the interpreter, without having to make changes to your program until you're more sure that you've found library functions that will help. For the most part, this kind of experimentation is harmless — if things don't work the way you expect, you'll see an error message or behavior that doesn't match what you expect — though you do need to be a little bit careful calling functions that do potentially destructive things like deleting files.

A note about versioning

As you're reading through Python's documentation, be sure that you're reading the right version of it. Near the top of each page is a drop-down list that allows you to select a version. Be sure you're reading the documentation for version 3.10 — documentation for any version beginning with "3.10" (e.g., 3.10.1, 3.10.2rc1, etc.) is fine. (In particular, if search engines lead you to Python documentation, you'll find that they quite often lead you to the documentation for older or newer versions of Python than 3.10. But you can usually just select "3.10" in the drop-down list in order to bring up the corresponding page of the documentation for our version of Python.)

Suggestions and limitations on what you can use

For the most part, if you find a function or type that can assist in your solution, you can feel free to use it; note that some care is sometimes required in ensuring that the function you find is actually a good fit. A careful reading of the documentation — and sometimes some reasoned experimentation — are a good way to know whether something might be suitable, but sometimes you won't realize that something isn't a good fit until after you've tried it. And that's fine; no one — no matter how experienced — makes the right design decisions every time.

You should consider looking at the **pathlib** library, which has tools for storing and manipulating paths in a way that is, more or less, portable between operating systems. Since one of your requirements is to support any operating system — as opposed to just one — you'll want to use the **Path** type in the **pathlib** library to represent paths. We'll see more about how to do that in lecture, but it's definitely the case that you'll want to avoid simply using strings for this purpose.

There are a few functions in the Python Standard Library that are off-limits for your use in this project. In general, you are required to write your own recursive function to search the filesystem (i.e., a function that searches in one directory, then recursively searches its subdirectories), which is one of the skills I'd like you to build in this project. That rules out anything that does a complete search in a single function call. For example, **os.walk** and **os.fwalk** — both of which perform a full traversal of files in a directory structure — fall into this category, along with the **glob** and **rglob** methods of the **Path** type (and probably others that I've not listed here). If you find yourself solving the problem of finding files without writing a recursive function to traverse them, you've used something you shouldn't.

(Note that, in general, it's safe to assume that you can use anything I don't specifically call out as being off-limits. I do sometimes leave certain things off-limits, mainly so that you achieve the learning objectives of the project; in this case, one of the key objectives is learning about recursion.)

The value of working incrementally

You may be accustomed to solving relatively small, mostly self-contained problems that you can handle all at once. This program, while not giant by real-world standards, consists of more moving parts than you might be used to writing, so you will likely find that attempting to write this program all at once will lead you astray, even if an all-inclusive, everything-at-once approach worked well for you in ICS 31 or past coursework.

A better approach for this project — and one that will increase in importance this quarter, as the problems we solve get larger and more complex — is to look for stable ground as often as you can find it. Rather than trying to write the entire program, write some small part of it that you understand well, then find a way to verify that the part you wrote works as you expected (e.g., by writing a function and then calling it in the Python interpreter manually). When it does, you've reached stable ground, and you're ready to choose the next small step you should take, again taking care to choose something that you'll be able to verify after you're done with it. Ideally, you'll find yourself reaching stable ground quite often — sometimes, every few minutes, if things are going well — and this will help you to feel confident that you're making progress.

If you find yourself stuck on one problem and have no idea how to move further, find another positive step you can take. For example, if you're not sure how to find all of the files in a directory structure, write a function that simply returns a hard-coded list of file paths and use that temporarily, then move on and work on something else, eventually working your way back to the places that were causing you trouble. The goal is always to be making some kind of progress, and it's not necessary to write the program in a particular order (e.g., the order in which things happen in the user interface).

Also, when you reach what you believe to be stable ground, it's not a bad idea to make a copy of your Python module before proceeding. That way, if your next step doesn't go as you planned, you maintain the option of "rolling back" to the previous, stable version and trying again. It also gives you something stable to turn in if you find that the deadline arrives and you're not finished; it's always better to turn in a partial program that does *something* correctly, rather than one that doesn't run. Don't feel like you need to keep *every* stable version, but it's not a bad idea to at least have the most recent one or two. (One of the practical skills you'll need to start thinking about, if you haven't already, is staying organized. If you have a couple of versions of a file and find that you're often not sure which is which, then you need a better organizational scheme — better file names, better directory names, or whatever helps it to be more obvious to you.)

So, what steps should I take?

There are a variety of ways to build this program from beginning to end, so don't go looking for the "perfect way" to do it. Find a small step you can take, take it, and then find another. Of course, there are missteps you might take along the way, but the best way to learn how to write programs this way is simply to do it; you'll learn as much from your missteps as you will from the ones that work out well.

As an example, though, think about the fact that the program is built around the core notion of "Find all of the unique files in a directory, its subdirectories, their subdirectories, and so on, and return a list of their paths." That sounds like a pretty good step to start with, except that it's actually a bigger step than it sounds like. So you could start even smaller: write a function that finds the files in a directory but ignores subdirectories. Once you can do that, add handling for one level of subdirectories (but assume they have no subdirectories inside of them). Then consider unbounded subdirectory depth and handle that scenario.

Whenever you're working on something and you feel you've bitten off more than you can chew, think about ways to break the problem into smaller ones; eventually, you'll be left with a problem you can think through and complete.

If you're not sure what step to take next, feel free to ask us; we'll help you find a task that will lead you to stable ground.

Sorting and lexicographical ordering

Sorting with key functions

When you want to sort numbers, there is a natural and obvious way that they should be ordered. If you don't say otherwise, the numbers will be sorted into *ascending order*, which means that smaller numbers would appear in the sorted order before larger ones. So, for example, if you sorted the list **[4, 2, 7, 5, 1, 8, 6, 3]** in a Python shell, it would become **[1, 2, 3, 4, 5, 6, 7, 8]**.

```
>>> x = [4, 2, 7, 5, 1, 8, 6, 3]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5, 6, 7, 8]
```

We would say, then, that numbers have a *natural ordering*, which is that they be ordered in an ascending fashion — smaller ones before larger ones.

That's not to say that you can't sort them in some other way. For example, if you wanted to sort the numbers into *descending order* instead, you can pass a *key function* to the **sort** method, whose job is to generate a *sort key* for each value in the list. The values will then be sorted on the order of their sort keys, rather than the values themselves. For example, suppose you had this function:

```
def negate(n: 'number') -> 'number':
    return -n
```

If you passed that function as **sort**'s key function, then you would be sorting the values on the basis of their negations. This would have the effect of making bigger numbers appear smaller and smaller numbers appear bigger, which would cause the elements to be sorted in the reverse of the order they'd appear by default — so they'd end up in descending, rather than ascending, order.

```
>>> x = [4, 2, 7, 5, 1, 8, 6, 3]
>>> x.sort(key = negate)
>>> x
[8, 7, 6, 5, 4, 3, 2, 1]
```

You can use that same technique to sort values that aren't numbers, provided that you have a key function that can generate a sort key for each one. For example, you could sort strings based on their lengths — with shorter ones appearing before longer ones in the sorted order — by using the built-in **len** function as the key function:

```
>>> x = ['Boo', 'is', 'happy', 'this', 'afternoon']
>>> x.sort(key = len)
>>> x
['is', 'Boo', 'this', 'happy', 'afternoon']
```

So, generally, we would say that **list**'s **sort** method works like this:

- If no key function is provided, sort the values by their natural ordering. (For example, numbers are sorted into ascending order.)
- If a key function is provided, sort the values using the natural ordering of their sort keys instead.

Lexicographical ordering: The natural ordering of strings

It's rare that someone is surprised when they learn that the natural ordering of numbers is ascending, because that tends to be the way they've learned about sorting numbers their whole lives. But what is the natural ordering of values that aren't numeric? The answer depends on what types of values we're talking about. Because it's germane to this project, let's focus our attention on the natural ordering of strings. First thing's first: Let's try an example and see what happens.

```
>>> x = ['f', 'b', 'a', 'h', 'c', 'g', 'd', 'e']
>>> x.sort()
>>> x
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Having sorted a bunch of one-character strings, each containing a single lowercase letter, the strings were sorted in alphabetical order. This seems somewhat natural for a lot of people, because a lot of us have been sorting strings alphabetically our whole lives, too. But are strings really sorted alphabetically? Let's try another example, where some of the letters are uppercase and others are lowercase.

```
>>> x = ['F', 'b', 'A', 'h', 'C', 'g', 'D', 'e']
>>> x.sort()
>>> x
['A', 'C', 'D', 'F', 'b', 'e', 'g', 'h']
```

That's a somewhat more curious result. The strings weren't sorted entirely alphabetically this time. The uppercase letters all appear before any of the lowercase letters, but the uppercase letters are sorted alphabetically with respect to one another and so are the lowercase letters. That seems like a strange rule, though; why should uppercase letters be treated as completely separate from lowercase ones?

What about strings of varying length? What happens with those?

```
>>> x = ['alex', 'hello', 'boo', 'alexander', 'booboo', 'bio']
>>> x.sort()
>>> x
['alex', 'alexander', 'bio', 'boo', 'booboo', 'hello']
```

This is more like the standard ordering you might see in a printed dictionary or in the index at the back of a book, where words are *alphabetized*. When comparing two words, the following algorithm is applied:

- If the first letter of one is earlier in the alphabet than another, it appears earlier in the sorted order.
- If the first letter of one is the same as another, we proceed to comparing the second letter in the same way to differentiate. If the second letters are the same, we proceed to the third, and so on. If all of the letters of one are the same as the initial letters of another, the shorter string appears first (which explains why **'boo'** is appearing before **'booboo'** and **'alex'** before **'alexander'** in the example above).

But things aren't quite so clear when we mix in uppercase letters again.

```
>>> x = ['alex', 'hello', 'Boo', 'Alexander', 'booboo', 'bio']
>>> x.sort()
>>> x
['Alexander', 'Boo', 'alex', 'bio', 'booboo', 'hello']
```

It turns out that there's an algorithm that explains all of this, which is called *lexicographical ordering*. The lexicographical ordering of strings works almost like the alphabetical ordering you may have seen in printed books, but there is a key difference. It centers around the idea that we can take two characters and decide which one comes first. We'll say each character has a numeric **ord** value, so a character comes before another if its **ord** value is smaller.

- Compare the first character of the two strings. If one of them has an **ord** value smaller than the other, it comes first.
- If the first character of each string is the same, do the same with the second character. Continue forward in this way until you find a character that's different, in which case the string with the smaller **ord** value comes first.
- If one of the strings runs out of characters before the other (e.g., we're comparing **'boo'** and **'booboo'**), the shorter string comes first.

Now the only remaining question is what the **ord** value of a character is. Python provides a built-in function called **ord** that can tell you this for any single-character string.

```
>>> ord('A')
65
>>> ord('B')
66
>>> ord('Z')
90
>>> ord('a')
97
>>> ord('b')
98
>>> ord('z')
122
```

And so we see the reason why uppercase letters sort earlier than lowercase ones; their **ord** values are smaller. The **ord** values are not arbitrary, as it turns out. Python uses a *character set* called Unicode, which specifies a numeric code for every possible character, including not just English letters, but also a variety of letters in other printed languages, along with symbols, emojis, and so on. Each of these characters has a code associated with it, and it's this code that is returned by **ord**. It turns out that Unicode specifies that uppercase English letters all have codes smaller than any lowercase English letters, so that explains some of the strangeness we were seeing before; it's not so strange after all. And we can find the **ord** value for any character this way:

```
>>> ord('>')
62
>>> ord('}')
125
>>> ord('⇒')
8658
```

So all you need to do if you want to understand the natural ordering of a sequence of strings is to find out the **ord** values for each of the characters of that string. In general, smaller **ord** values come before larger ones. It's really that simple.

Testing

Testing this program is going to seem cumbersome, because it will require creating directory structures and files in various configurations, then running the program to see how it behaves. You might find that it's worth your time to automate some scenarios, by writing short Python functions (perhaps in a separate module) that create directories and files in interesting combinations. You won't likely find that it will require writing a lot of code, but it will pay you back (and then some!) as you test your program.

It's quite common in real-world software development to write code whose sole use is as a development tool; it's not part of the program, per se, and will not be given to users of the program, but is strictly meant to make it easier to build the program. You'll be well-served to explore ways to use Python to lighten your testing burden; if there are five interesting scenarios you've identified, you should write five Python functions that can set them up for you automatically, so you can get them back any time you need to test them. If it takes a half-hour to write the test programs, but it takes five minutes to set up the tests manually, as soon as you've set up the tests six times, the time spent writing the test program will begin paying you back. Computers automate tasks that are otherwise cumbersome, and testing certainly falls into that category. (We'll see this theme repeated throughout the course.)

Sanity-checking your output

We are also providing a tool that you can use to sanity-check whether you've followed some of the basic requirements above. It will only give you a "passing" result in these circumstances:

- You've named your file **project1.py**.
- Executing that module is enough to execute your program.
- Your program reads its input and generates character-for-character correct input for one test scenario, which is similar to the example inputs and outputs shown above.

Note that many additional test inputs will be used when we grade your project, as there are a number of combinations — searching recursively or not, multiple search characteristics, multiple acitons — that are possible. The way to understand the sanity checker's output is to think of it this way: Just because the sanity checker says your program passes doesn't mean it's close to perfect, but if you *cannot* get the sanity checker to report that your program passes, it surely will not pass all of our automated tests.

Running the sanity checker is simple. First, download the Python module linked below:

- project1_sanitycheck.py

Put that file into the same directory as your **project1.py** file. Running the **project1_sanitycheck.py** module — for example, by loading it in IDLE and pressing F5 (or selecting **Run Module** from the **Run** menu) — will run the sanity checker and report a result, which will be printed to the Python shell.

Note, too, that this program's output depends not only on the program's input, but also on the presence of files and directories. For this reason, every time you run the sanity checker, it will create a test directory — a subdirectory in the same directory as **project1_sanitycheck.py** — and place some files into it that have the appropriate characteristics. So that you can understand its output better, the sanity checker will leave that subdirectory behind, though you can, of course, feel free to delete it when you're done with it.

Deliverables

Put your name and student ID in a comment at the top of your one and only **.py** file (**project1.py**), then submit the file *and only that file* to Canvas.

Be aware that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

What do I do if Canvas slightly adjusts my filename?

Canvas will sometimes modify your filenames when you submit them (e.g., when you submit the same file twice, it will change the name of your second submission to end in **-1.py** instead of just **.py**). In general, this is fine; as long as the file you submitted has the correct name, we'll be able to obtain it with that same name, even if Canvas adjusts it.