ICS 32 Fall 2016
Project #1: *Begin the Begin*

Due date and time: *Friday, October 7, 11:59pm*

This project is to be done in pairs using the "pair programming" technique

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).

Choosing a partner

During your lab section on Monday, September 26, choose a partner from among the students who are *officially enrolled in the same lab section as you*. It's fine, even preferable, to read this project write-up on your own ahead of time, though, so you and your partner can hit the ground running when you start working together, but do not start working on your solution until you are partnered up. (We understand that you might be eager, but the goal here is to take a shared journey with someone else, not to arrive on the first day and say "Okay, we're already done!")

Once you've selected a partner, notify your TA of your partnership during the lab section. Your TA will approve the partnership and make a note of it, at which time you're officially partners! (Until you've received this approval, you are not yet partners.)

For those of you who are unable to attend lab on Monday, September 26, there's a backup plan: your TA will randomly select partnerships from among the students remaining after the section is over, and will notify you and your new partner via email. Once your TA has selected a partner for you in this fashion, we will not allow you to switch to another one, so the best way to control your destiny is to choose a partner yourself during your lab section on Monday, September 26.

If you're having trouble finding a partner, notify your TA during your lab section, so that you can be assisted in finding one.

You will not receive credit on this project if you work on it alone, without the prior consent of the instructor. (Please note that "prior consent" does not include approaching us the day the project is due, having completed it on your own, and telling us that you haven't been able to find a partner. Pair programming is a part of the course, and we expect everyone to participate in it when asked.)

Be sure, too, that you've read the section entitled *Pair programming* on the front page of the Project Guide, so you're aware of what it means to participate in pair programming.

Reusing your own work from a previous quarter

Be aware that if you've taken this course previously — or if you've partnered with someone who has — the previous quarter's code cannot be used, in whole or in part, during your work on the project this quarter, as we expect partnerships to do their own work; if one partner brings a partial or complete solution to the table, the other partner is deprived of the journey of solving the problem (which is the objective of these projects). This rule is detailed more thoroughly, along with other academic honesty rules, on the front page of the Project Guide, but you should be aware that there are ramifications for *both partners* if you reuse code from a previous iteration of the course when partnering.

Your best move, in this situation, is not to refer to your previous quarter's work at all. (Starting with the previous quarter's work and modifying it to look "different enough" is not doing original work; you are required to start fresh.) This is generally a good practice when retaking a course, anyway, because the reason you're retaking the course is because there are things you didn't learn well enough previously in order to pass it. But it's especially important when working with a partner.

What to do if you're not officially enrolled in the course

If you have not yet officially enrolled in the course, you'll need to wait until you've enrolled before you partner up. However, you can feel free to work alone for the time being.

File systems

Whether you run Windows, Mac OS X, 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 Assignment #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 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 search for files in a directory (and all of its subdirectories, and their subdirectories, and so on) with interesting characteristics and take some kind of action on those files. Both the notion of *interesting characteristics* and *taking action* will be configurable and can each 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 problem, 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; it should simply read whatever input is typed into the console, and you should assume that your user knows the precise input format. A description of that format follows.

- The first line of the input is the path to the directory in which the search for files should be rooted. For example, a Windows user might type **C:\Program Files** or **D:\Python34\Lib**, in which case only files in or "underneath" the chosen directory will be found.
    - If the first line of the input is not a valid path to a directory (e.g., not a path to a directory that exists, or is not a valid path at all), print the word **ERROR** on a line by itself and repeat reading a line of input; continue until the input is a valid path to a directory.
- The second line of the input specifies the *search characteristics* that will be used in deciding whether files are "interesting" and should have action taken on them. There are three different search characteristics; the second line of the input chooses one of them.
    - If the second line of the 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 space; after the space, the rest of the line will indicate the name of the files to be searched for.
    - If the second line of the input begins with the letter **E**, the search will be for files whose name end in 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 the second line of the input begins with the letter **S**, the search will be for files whose size, measured in bytes, strictly exceeds (i.e., is greater than) a specified threshold. The **S** 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 **S 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 the second line of the 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.
- The third line of the input specifies the *action* that should be taken on each of the interesting files found in the search. No matter what, you should always print the file's path, on its own line of output, to the console when you find an interesting one; the action chosen here specifies what else should be done with it.
    - If the third line of the input contains the letter **P** by itself, print the file's path to the console — just as you will always do — but otherwise don't do anything with it.
    - If the third line of the input contains the letter **F** by itself, open the file under the assumption that the file is a text read, read the first line of text from the file, and print that text to the console. (If the file does not contain text, it's fine if this choice prints unreadable garbage or even crashes your program; testing that a file is a text file is trickier than it sounds, so we'll only test this feature on text files.)
    - If the third line of the 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 the filename. For example, if the interesting file is **C:\pictures\boo.jpg**, you would copy it to **C:\pictures\boo.jpg.dup**.
- If the third line of the 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 the third line of the 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.

Running the search and taking action

Your search should look for files in the directory where the user asked for the search to be rooted, in any subdirectories of that directory, in any of their subdirectories, and so on, for as deep as the directory structure goes.

Since one of the goals of this project is to introduce you to the use of recursion to solve real problems, your search 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").

In general, the order in which you take action on files is not important, so long as every interesting file has the action taken on it exactly once.

Once the three lines of input are read and a single search is completed, the program ends.

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.

```
D:\Test\Project1\Example
Q
ERROR
S boo
ERROR
S 1234
F
D:\Test\Project1\Example\test1.txt
This is a line of text
D:\Test\Project1\Example\Sub\meee.txt
Hello, my name is Boo
D:\Test\Project1\Example\Zzz\zzz.py
def this_function():
```

Portability

It should be possible to run your program on any operating system — Windows, Mac OS X, Linux, etc. — so long as there is a Python 3.5 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 Mac OS X, 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 note the failure (by printing a readable message to the console) 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 one scenario where it's fine for your program to misbehave (or even crash) is when printing the first line of text from a file that is not a text file (e.g., an image or a video). It turns out that detecting whether a file contains only text is not nearly as trivial as it sounds, so we'll avoid that problem here.

Organization of your program

Your program should be written in a single Python module (i.e., a single **.py** file). You can feel free to name the file anything you'd like, but running your module (e.g., by loading it in IDLE and pressing F5 or selecting **Run Module** from the **Run** menu) should result in your program being executed and your user interface being displayed, but importing your module manually using an **import** statement should not. 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).

We'll see examples of this technique in lecture, and we'll tend to use it throughout the quarter, so you'll want to be sure you get this part right.

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: [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.

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 3.5 documentation
- Python 3.5 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**, **os.path**, 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 3.5 — documentation for any version beginning with "3.5" (e.g., 3.5.1, 3.5.2rc1, 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 versions of Python than 3.5, which is relatively new. But you can usually just select "3.5" 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.

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.

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.)

Deliverables

Only one of the two partners should submit the project; we are aware of the partnerships, so we will be able to figure out which project submissions belong to which pairing. Put the names and student IDs of both partners in a comment at the top of your one and only **.py** file, then submit the file to Checkmate.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want greaded. 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.

Communicating about your partnership

Partner evaluation survey

After you've completed your project and submitted it, please complete a *partner evaluation survey*, which you will find on EEE around the time the project is due. (If you don't see it, check again nearer the due date.) This gives you an opportunity to provide us with honest feedback — the good and the bad — about working with your partner.

What to do if your partner is absent or uncooperative

The goal here is to do *pair programming*, which is described in more detail on the front page of the Project Guide. If your partner is consistently absent, uncommunicative, or uncooperative when it comes to pair programming, you can feel free to contact me (the instructor) and I will be able to arbitrate the matter. Single absences from lab happen — though you should have the decency to contact your partner, e.g., via email or a text message if you're not going to be able to make it — so I don't need to kept aware of one-off issues like that, but I do want to know about consistent issues. We do expect everyone to participate in pair programming when assigned, and we reserve the right to reduce scores (and overall course grades) of students who persistently refuse to do so.

More clarification on pair programming added by Alex Thornton, Spring 2015.
Input format overhauled by Alex Thornton, Winter 2015.
Updated to include Python 3.4 features by Alex Thornton, Fall 2014.
Clarification on off-limits Python features added by Alex Thornton, Spring 2014.
Clarification on rules regarding reuse of prior work added by Alex Thornton, Winter 2014.
A few additional knobs turned by Alex Thornton, Fall 2013.
Some tweaks and clarifications by Alex Thornton, Spring 2013.
Originally written by Alex Thornton, Winter 2013.