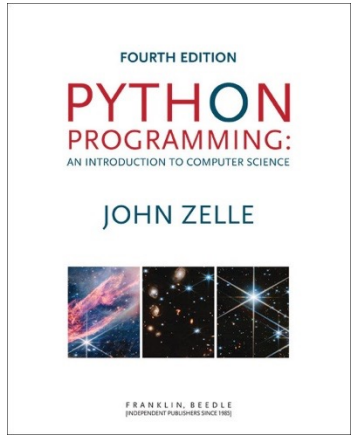
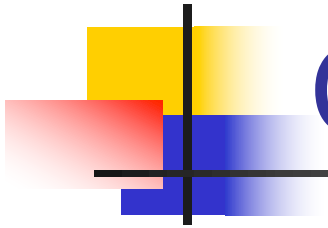
Python Programming: An Introduction To Computer Science

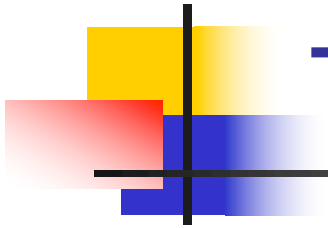


Chapter 10
Persistent Data



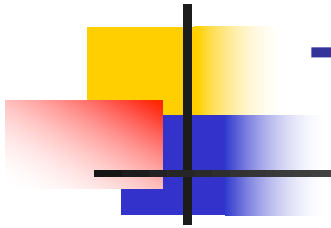
Objectives

- To understand basic file-processing concepts and techniques for opening, reading, and writing files in Python.
- To understand the structure of text files and be able to write programs that use them.
- To become familiar with the basic organization of file systems, including role of absolute and relative paths play in locating files, and be able to write Python programs that process collections of files.



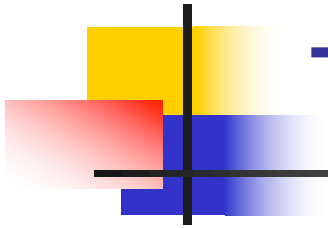
Text Files

- In all of the examples so far, data has either been embedded in the program code or entered by the user when the program runs.
- We lack a mechanism for entering data and having that data persist from one run of the program to the next.



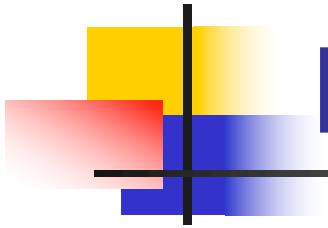
Text Files

- Persistent data is a critical component of any modern computing system.
  - Your word processor needs to save the paper you're working on.
  - Your programming environment needs to be able to save and reload your Python code.
- Typically, such information is stored in files.



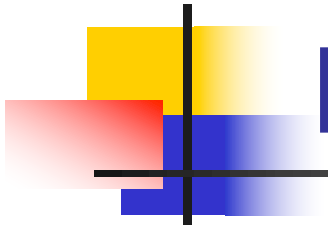
Text Files

- A *file* is a sequence of data that is stored in secondary memory (usually on a disk drive of some sort).
  - Files can contain any data type, but the easiest files to work with a those that contain text.
  - Files of text have the advantage that they can be read and understood by humans, and they are easily created and edited using general purpose text editors, like IDLE.



Multi-line Strings

- You can think of a text file as a (possibly long) string that happens to be stored on disk.
- A special character or sequence of characters is used to mark the end of each line.
  - While this convention varies by operating system, Python takes care of these different conventions for us and just uses the regular newline character (`\n`).



Multi-line Strings

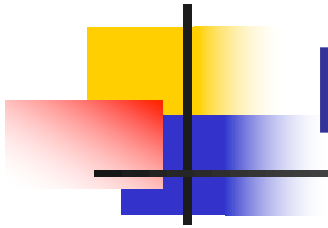
```
Hello
World


Goodbye 32
```

- When stored to a file, you get this:

```
Hello\nWorld\n\nGoodbye 32\n
```

  - Notice that the blank line becomes a bare newline.



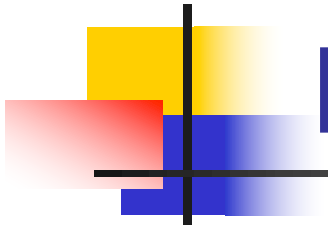
Multi-line Strings

- This is no different than when we embed newline characters into output strings to produce multiple lines of output with a single `print` statement.

```
print("Hello\nWorld\n\nGoodbye 32\n")
```

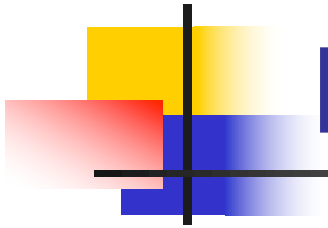
- Remember, if you simply evaluate a string containing newline characters in the shell, you will just get the embedded newline representation back.

```
"Hello\nWorld\n\nGoodbye 32\n"
```



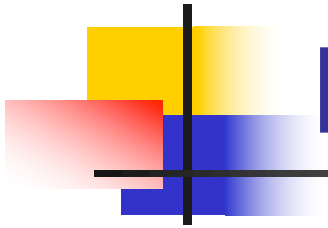
File Processing Outline

- Virtually all programming languages share certain underlying file manipulation concepts.
  - We need some way to associate a file on disk with an object in a program – this is called *opening* a file.
  - We need a set of operations that can manipulate the file object.
    - At the very least, we need to be able to read the information from a file and to write new information to a file.
  - Lastly, when a we are done we need to *close* the file.



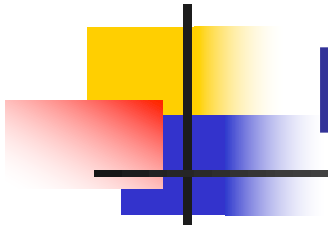
File Processing Outline

- This idea of opening and closing files is closely related to how you might work with files in an application program such as IDLE.
  - When you open a file for editing in IDLE, the file is actually read from disk and stored in RAM.
  - At this point, the file is closed (in the programming sense).
  - As you edit the file, you are really making changes to the data in memory, not the file itself.
  - Changes will not show up on disk until you "save" it.



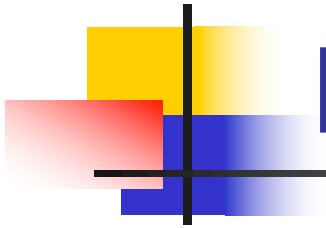
File Processing Outline

- The process of saving a file in IDLE is also a multi-step process.
  - The original file on the disk is opened, this time in a mode that allows it to store information (opened for *writing*).
  - Doing this actually *erases* the old contents of the file!
  - File writing operations are then used to copy the current contents of the in-memory file into the new file on disk.



File Processing Outline

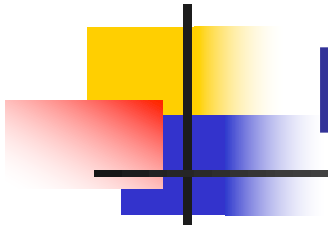
- Working with text files in Python is easy!
  - Create a file object that corresponds to a file on disk:
    `<variable> = open(<path>, <mode>)`
  - Here, `path` is a string that provides the location of the file on disk.
  - For a text file, `mode` is either "r" or "w" depending on whether the file intended to be *read* from or *written* to.
  - If the mode is omitted, the file is opened for reading.



File Processing Outline

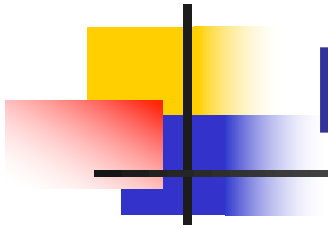
```
# printfile.py
#   Prints a file to the screen.

def main():
    fname = input("Enter a filename: ")
    infile = open(fname, "r")
    data = infile.read()
    infile.close()
    print(data)
```



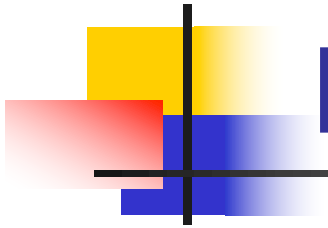
File Processing Outline

- The program first prompts the user for a file name and then opens the file for reading through the variable `infile`.
  - While any identifier works, here the name serves to remind us that the object is a file and it is being used for input.
- The entire contents of the file is then read as one multi-line string and stored in the variable `data`.
- Printing `data` causes the file contents to be displayed.



File Processing Outline

- This process illustrates the basic three-step process for working with a file:
  1. Open the file.
  2. Use file operations to read or write data.
  3. Close the file.
- Any file that is opened should be closed when the program is done using it. Technically, all files get closed when the program terminates, but doing it explicitly is good programming style.

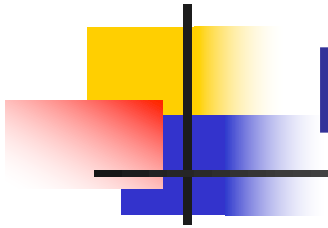


File Processing Outline

- In order to make sure that necessary actions such as closing a file occur, Python has a powerful feature called a *context manager*.

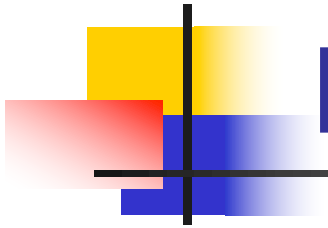
```
# printfile2.py
#   Prints a file to the screen.

def main():
    fname = input("Enter a filename: ")
    with open(fname, "r") as infile:
        data = infile.read()
    print(data)
```



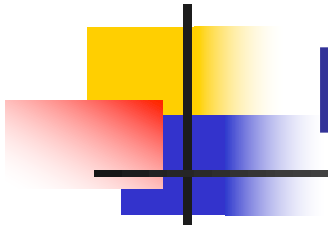
File Processing Outline

- The `with` statement associates the variable with the file object created by `open`.
- The file object acts as a context manager for executing the instructions in the indented body of the with.
- When the body has completed, the file will be closed automatically, even if control leaves the body due to an exception or `return` statement.



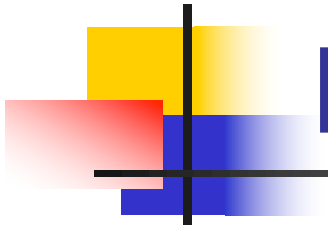
Reading from a File

- `read` is just one of several options that can be used to access the contents of a file.
  - `<file>.read()` – Returns the entire remaining contents of the file as a single (potentially large, multi-line) string.
  - `<file>.readline()` – Returns the next line of the file, i.e. all text up to *and including* the newline character.
  - `<file>.readlines()` – Returns a list of the remaining lines in the file. Each list item is a string of a single line including the newline character at the end.



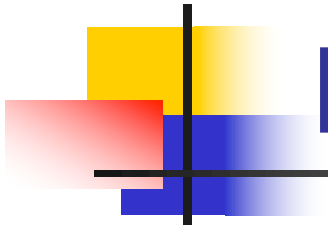
Reading from a File

- Text files are read sequentially – the system keeps track of what has been read since a file has been opened, so that a later read will pick up where the previous one left off.
- If you want to read a previous line, you need to close and reopen the file.



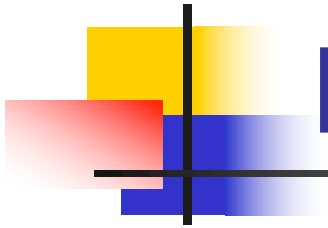
Reading from a File

- Successive calls to `readline()` read successive line from the file.
- The string returned by `readline()` will always end with a newline character.
- Use slicing to strip off the newline character at the end of the line, otherwise it will look double-spaced.
- Or, you could also tell print to not add its own newline, e.g. `print(line, end="")`.



Reading from a File

```
with open(someFile, "r") as infile:
    for _ in range(5):
        line = infile.readline()
        print(line[:-1])
```

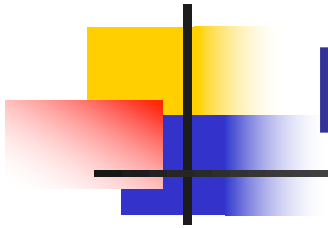


Reading from a File

- One way to loop through the entire contents of a file is to read in all of the file using `readlines`, then loop through the resulting list.

```
with open(someFile, "r") as infile:
    for line in infile.readline():
        # process the line here
```

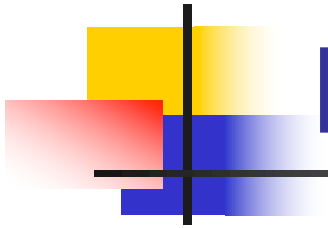
- What happens if the file is too large to fit in your computer's memory?



Reading from a File

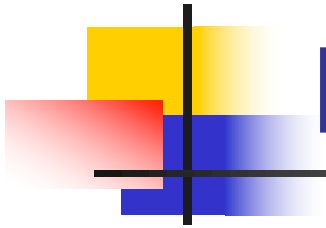
- Python treats a file as sequence of lines, so looping through the lines can be done directly:

```
with open(someFile, "r") as infile:
    for line in infile:
        # process the line here
```



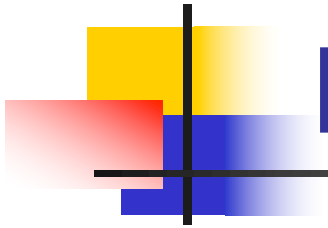
Reading from a File

- Let's improve our statistics library from last chapter.
- One disadvantage of the previous version is that `getNumbers()` gets numbers from the user interactively.
- What if you are trying to average one hundred numbers and you make a mistake on number 98? Doh! You'd need to start over again.



Reading from a File

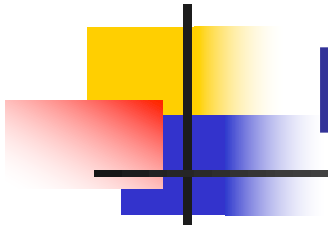
- A better approach – type all the numbers into a file. We can then edit the data before sending it to the program.
- This file-oriented approach is typically used for data-processing applications.
- We can improve the usefulness of our library by adding a `getNumbersFromFile` function that takes the name of a file as a parameter and returns a list of numbers read from the file.



Reading from a File

- Suppose our numbers are in a text file, with each line containing a single number.

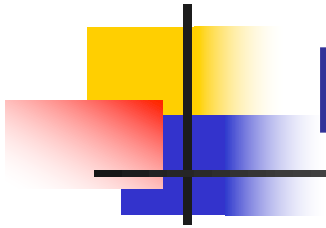
```
def getNumbersFromFile(fname):
    nums = []
    with open(fname, "r") as infile:
        for line in infile:
            nums.append(float(line))
    return nums
```



Reading from a File

- We could also do this more succinctly with a list comprehension:

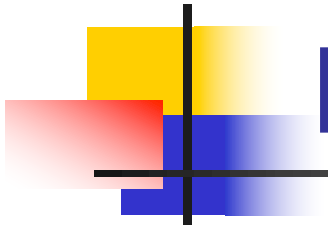
```
def getNumbersFromFile(fname):
    nums = []
    with open(fname, "r") as infile:
        nums = [float(line) for line in infile]
    return nums
```



Reading from a File

- Using this approach, we need to be very careful with the format of the input file – there must be *exactly one* number on each line.
- A common error is to introduce an extra blank line at the bottom that may go unnoticed. This would cause

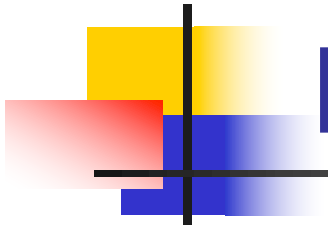
```
in <listcomp>
nums = [float(line) for line in infile]
ValueError: could not convert string to float: ''
```



Reading from a File

- We could make our function more flexible by having it accept multiple numbers on the same line.
- A single line can easily be turned into a list of numbers using split in the list comprehension, similar to what we did when we had multiple numbers on a single line of interactive input:

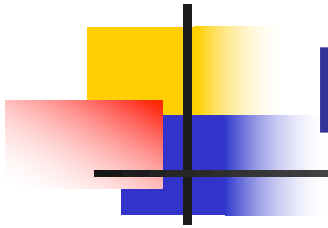
```
nums = [float(num) for x in line.split()]
```



Reading from a File

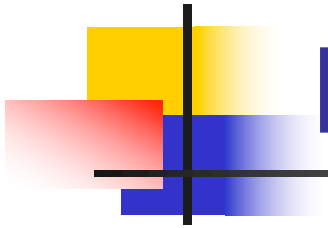
- To get all the numbers across multiple lines, we simply wrap this up in an accumulator loop that processes the lines of the input file:

```
def getNumbersFromFile(fname):
    nums = []
    with open(fname, "r") as infile:
        for line in infile:
            newnums = [float(num) for x in line.split()]
            nums.extend(newnums)
    return nums
```



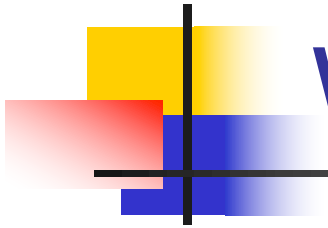
Reading from a File

- Here the accumulator is called `nums` and the list created from each line is called `newnums`.
- The final line in the loop body appends the numbers from the current line to the end of the accumulator using the `list` extend method introduced in chapter 9.
- This version of the stats program appears in `stats3.py`.



Reading from a File

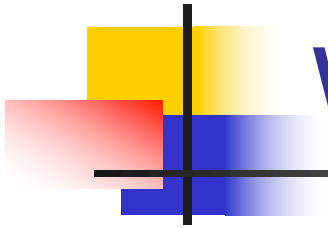
- Using this approach has several benefits:
  - It allows you to create a data file with as many numbers on each line as you want.
  - The program will also be more robust by handling accidental blank lines (Do you see how?).



Writing to a File

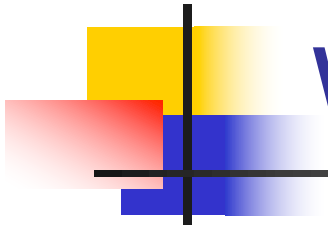
- Opening a file for writing prepares that file to receive data.
- If no file with the given name exists, a new file will be created.
- **If a file with the given name *does* exist, Python will delete it and create a new, empty file.**

```
with open("mydata.out", "w") as outfile:
    # do things with outfile here
```



Writing to a File

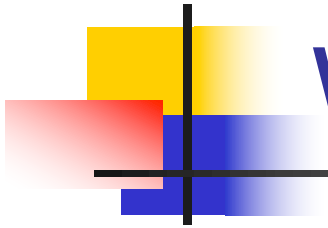
- The easiest way to write information into a text file is to use the `print` function.
- To do this, simply add an extra keyword parameter that specifies the file:
  `print(..., file=<outputfile>)`
- This behaves exactly like a normal `print`, except the result is sent to `outputfile` rather than the screen.



Writing to a File

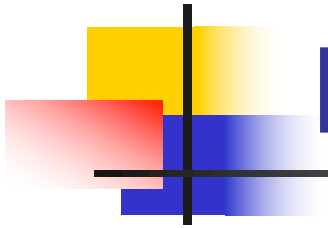
- Here's a program to create a text file with a haiku about programming:

```
# haiku.py
def main():
    haiku = ["White space and syntax",
             "Python code flows like water",
             "Solutions emerge"]
    print("I have a haiku for you.")
```



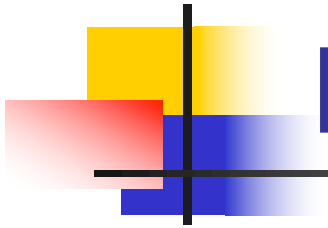
Writing to a File

```
fname = input("Enter a file name to receive the haiku: ")
with open(fname, "w") as haikufile:
    for line in haiku:
        print(line, file=haikufile)
print(f"Look in {fname} to see your haiku")
```



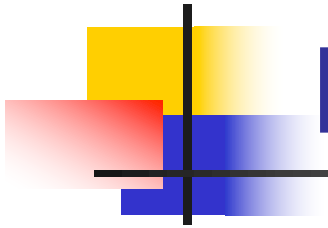
Batch Processing

- To see how these pieces fit together in a larger example, let's redo the username generation program from Chapter 8.
- Our previous version created usernames interactively by having the user type in his or her name.
- If we were setting up accounts for a large number of users, this process would probably not be done interactively, but in *batch* mode, where program input and output is done through files.



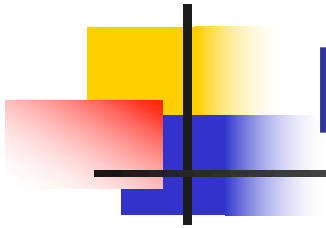
Batch Processing

- Each line of the input file will contain the first and last names of a new user separated by one or more spaces.
- The program produces an output file containing a line for each generated username.



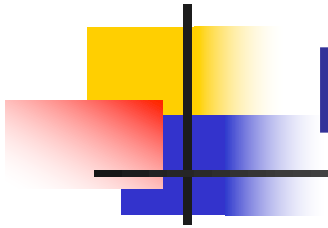
Batch Processing

```
# userfile.py
# Program to create a file of usernames in batch mode.
def main():
    print("This program creates a file of usernames from a")
    print("file of names.")
    # get the file names
    infileName = input("What file are the names in? ")
    outfileName = input("What file should the usernames go in? ")
```



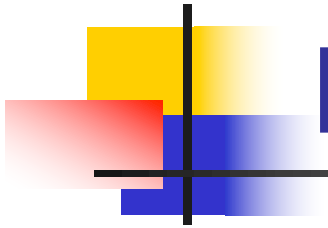
Batch Processing

```
    # open the files
    with open(infileName, "r") as infile, open(outfileName, "w") as outfile:
    # process each line of the input file
    for line in infile:
        # get the first and last names from line
        first, last = line.split()
        # create the username
        uname = (first[0]+last[:7]).lower()
        # write it to the output file
        print(uname, file=outfile)
print("Usernames have been written to", outfileName)
```



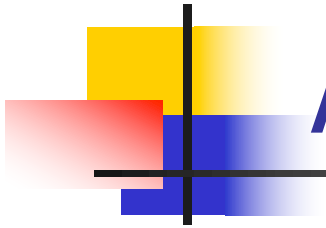
Batch Processing

- A couple things worth noticing:
  - Two files are open at the same time, one for input (`infile`) and one for output (`outfile`). This is accomplished in the `with` by including two `open(…) as <variable>` clauses separated by a comma. It's not unusual for a program to act on multiple files simultaneously.
  - When creating the username, the lower string method was used to ensure that the username is all lowercase, even if the input names are mixed case.



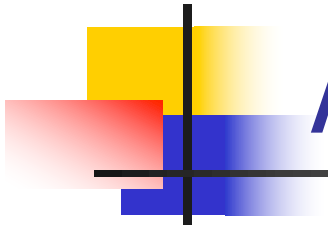
File Names and Paths

- So far in our examples we've indicated the file to be opened by supplying the name of the file as a string.
- Using this approach, files end up in the folder where the programs live.
- This might be OK for assignments, but in the real world we'd like users to be able to select files from anywhere in secondary memory.



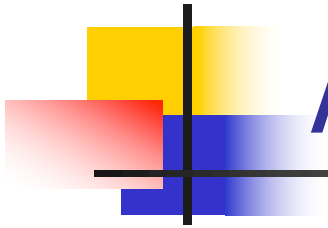
Absolute and Relative Paths

- Way back in Chapter 1 we looked at how a computer's operating system generally organizes secondary memory as a hierarchical collection of directories (also called folders) that can contain files as well as other directories.
- The directory at the top of this hierarchy is called the root directory.
- A file is located by specifying a *path* from the root directory down through the hierarchy of directories.



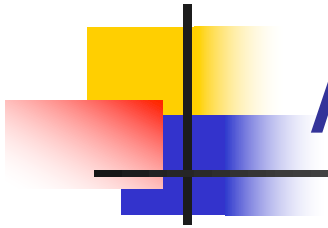
Absolute and Relative Paths

- E.g., the text of this chapter is in a file having the path `/home/zelle/Books/cs1book/cs1book4e/textbook/chapter10.tex`
- The top-level directory on Dr. Zelle's computer is designated with a `/`. His computer's root directory contains around 20 subdirectories, including one called `home`.
- A slash (`/`) is also used to separate the directory names along the path.



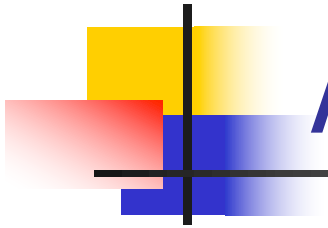
Absolute and Relative Paths

- You can think of the path from the root as representing the "full name" of any given file.
- The name has to be so complex because a typical computer contains millions of files; there must be a way to uniquely identify each of these files.
- This complete path to a given directory or file is called the *absolute path*.



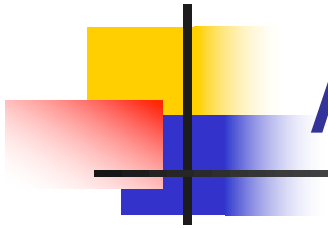
Absolute and Relative Paths

- Anywhere in Python where a file path is needed, an absolute path can be used.
- Working with absolute paths can be a pain!
  - They're long
  - Moving a file or folder changes the absolute paths of files and folders!
- Any path that beings with something other than the root directory is considered a *relative* path.



Absolute and Relative Paths

- When we just use the name of a file in our examples, those were relative paths.
- Running programs always have an associated *working directory* which is the directory that it is currently working in.
- Typically, this is the directory where your program file is located.

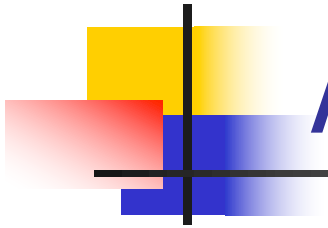


Absolute and Relative Paths

- Suppose we have a program `data_analyzer.py` stored in `/home/zelle/python`.
- When this program is run its working directory will be `/home/zelle/python`.

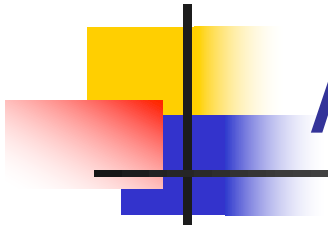
```
path = input("What file should I analyze? ")
with open(path, "r") as infile:
    # process the file
```

- If the user enters `nums.txt`, the program will look for `/home/zelle/python/nums.txt`.



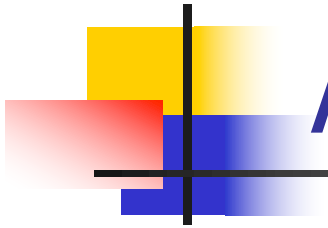
Absolute and Relative Paths

- Suppose the user instead enters `data/nums.txt`.
- Python will threat this as a path starting at the current working directory: **`/home/zelle/python/`**`data/nums.txt`.
- The characters "." and ".." have special meanings for relative paths.
  - "." indicates the current working directory
  - ".." indicates the parent of the current working directory.
  - In our previous example, an equivalent would be `../data/nums.txt`



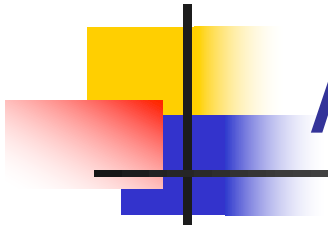
Absolute and Relative Paths

- Dr. Zelle's laptop is running Linux. While the ideas are the same, the details differ among operating systems.
- On macOS, a user's home directory is in `/Users`.
  - `/Users/zelle/data/nums.txt`
- On Windows, the path notation is a little different.
  - `C:\Users\zelle\data\nums.txt`
  - Each hard drive (`C:`, `D:`) has its own file system with its own root directory.
  - Windows uses `\` rather than `/` in paths



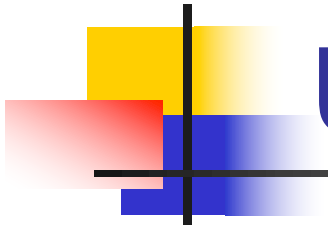
Absolute and Relative Paths

- Python always allows paths to be separated using a regular slash (`/`) on any OS for interoperability.
- It's best practice to avoid "`\`" in Windows paths in Python since the backslash is used in string literals to indicate special characters, i.e. `\t`, `\n`. To use an actual backslash in a literal, you'd need to escape it (`\\`) or prefix the string with r to indicate it is a "raw" string (don't interpret).



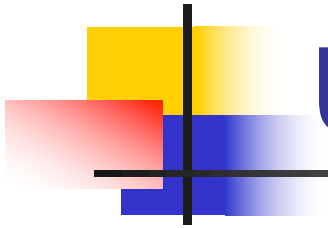
Absolute and Relative Paths

- Three ways to open the same file in Windows
  - `with open("data/nums.txt") as infile:  # generic Python`
    `# notation`
  - `with open("data\\nums.txt") as infile: # Windows notation`
    `# using special char`
  - `with open(r"data\nums.txt") as infile: # Windows notation`
    `# using raw string`
- The best one? Number one – it will work on other operating systems besides Windows.



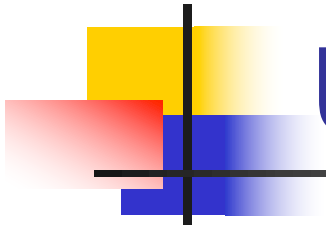
Using pathlib

- File are a ubiquitous part of the computing landscape, and just about every program has to manipulate them in one way or another.
- Python provides a library called `pathlib` to help with some of the common, but tedious tasks.
- The main tool is the `Path` object. `Path` is a sort of "wrapper" around a path string that gives it some convenient superpowers.



Using pathlib

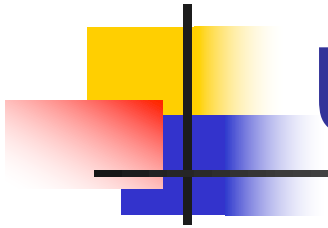
- Let's improve our batch-oriented username program so that it checks if the intended output file exists. If it does, create a backup of that file so that the contents aren't lost when the new usernames are written.



Using pathlib

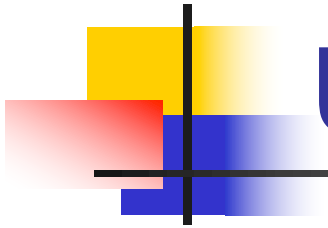
```
# userfile2.py
from pathlib import Path

def main():
    print("This program creates a file of usernames from a")
    print("file of names.")
    # get the file names
    inPath = Path(input("What file are the names in? "))
    outPath = Path(input("What file should the usernames go in? "))
```



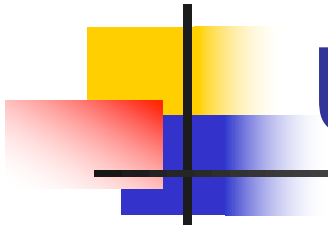
Using pathlib

```
# backup the output file if it already exists
if outPath.exists():
    backupPath = outPath.with_suffix(".bak")
    print(f"Renaming existing {outPath.name} to {backupPath.name}")
    outPath.rename(backupPath)
```



Using pathlib

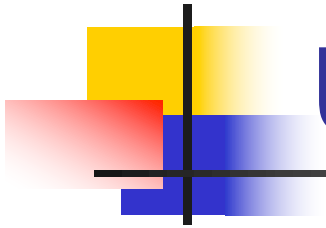
```
    # open the files
    with open(inPath, "r") as infile, open(outPath, "w") as outfile:
        # process each line of the input file
        for line in infile:
            # get the first and last names from line
            first, last = line.split()
            # create the username
            uname = (first[0]+last[:7]).lower()
            # write it to the output file
            print(uname, file=outfile)
print("Usernames have been written to", outPath)
```



Using pathlib

- You can extract different parts of a path using simple attributes from a Path object.

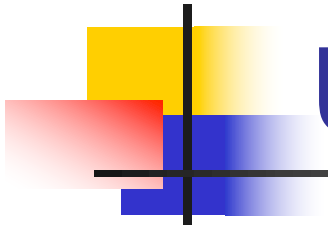
```
>>> path = Path("/home/zelle/python/data.txt")
>>> path.name
    'data.txt'
>>> path.stem
    'data'
>>> path.suffix
    '.txt'
```



Using pathlib

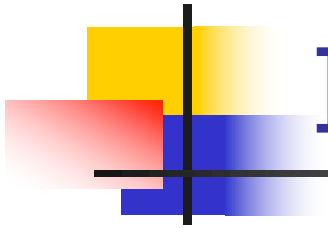
- We can create a slightly modified path by using `with_<part>` methods to replace specific parts in an existing path.
  - `backupPath = outPath.with_suffix(".bak")`
  - This creates a new `Path` that is just like `outPath`, except it has the extension (suffix) ".bak" instead of its original extension.
- Our program's output will look something like

```
Renaming existing usernames.txt to usernames.bak
```



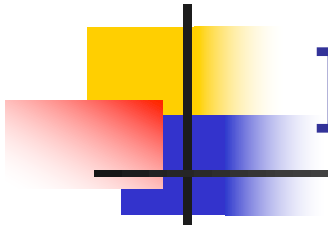
Using pathlib

- The actual renaming of the file happens with `outPath.rename(backupPath)`
- The rename method is one of a number of Path object methods that can be used to make changes in the underlying file system.
- The necessary commands differ by operating system, but the `Path` object handles the differences in a transparent way!



Iterating over Directories

- Another task that programs often need to do is to process a whole batch of files at a time.
- For example, a photo management app might allow the user to load all the images in a given directory.
- If you have a `Path` object that points to a directory on your hard disk, there are a couple methods that allow you to loop over the contents of that directory.

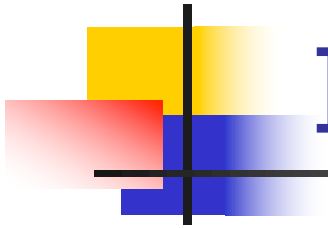


Iterating over Directories

- The simplest of these methods is `iterdir`.
- It produces a sequence of `Path` objects, one for each file or directory contained in the original directory.

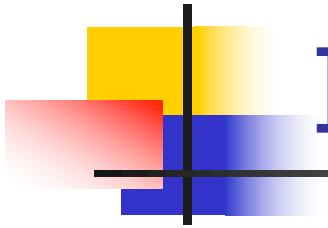
```
>>> path = Path(".")
>>> for p in path.iterdir():
            print(p)

names.txt
stats3.py
...
```



Iterating over Directories

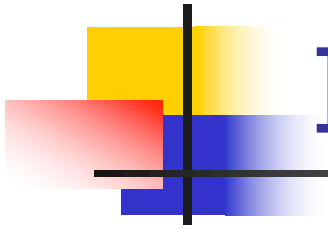
```
list(path.iterdir())
[PosixPath('names.txt'), PosixPath('test.txt'),
PosixPath('stats3.py'), PosixPath('data'),
PosixPath('nums1.txt'),
PosixPath('usernames.bak'), PosixPath('nums2.txt'),
PosixPath('usernames.txt'),
PosixPath('userfile2.py'),
PosixPath('userfile.py'), PosixPath('haiku.py')]
```



Iterating over Directories

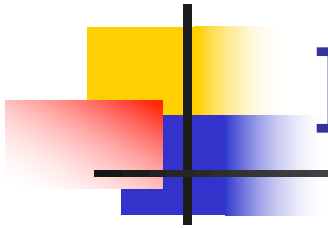
- Notice that each item in the sequence produce by `listdir()` is itself a `Path` object.
- It means we can make use of the various `Path` methods on these items.
- The `is_file` method returns `True` if the path is a file (as opposed to a directory).

```
files = [p for p in path.iterdir() if p.is_file()]
```



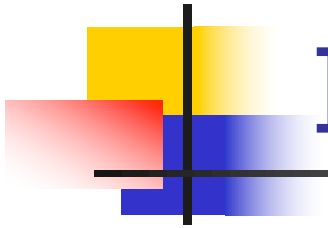
Iterating over Directories

- If we wanted just the Python program files, we could grab just the items that had a .py suffix.
- `python_files = [p for p in path.iterdir() if p.suffix == ".py"]`
- This last example could have been handled more simply using a technique known as *file globbing*.
- You can select a subset of files that match a pattern using the `glob` method:
  `path.glob(pattern)`



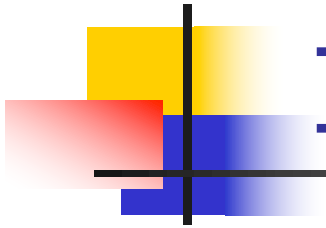
Iterating over Directories

- The pattern looks like a regular path string except that it can contain certain "wildcard" characters.
  - "?" matches any single character
  - "*" matches any sequence of characters
  - `python_files = list(path.glob("*.py"))`
  - The glob `"*.py"` will match any file that ends with .py



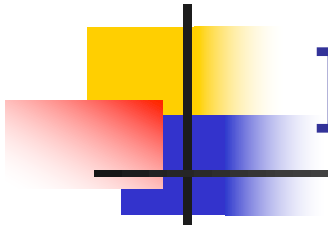
Iterating over Directories

- Our last addition was a `getNumbersFromFile(path)` function that can be used to get a data set from a specific file.
- Suppose we have a number of data sets, each stored in a separate file in our data directory.
- It would have handy to have a `getNumbersFromFiles` function making use of file globbing to accumulate all the data across the set of files.



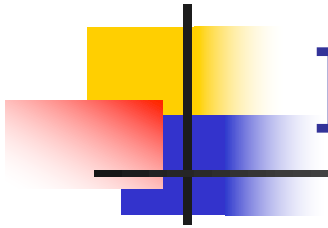
Iterating over Directories

- Let's write a function with two parameters.
  - `basedir` gives the directory containing the data
  - `pattern` is a pattern for which files to look in
  - To get the number from all the flies in a data directory, we could do `data = getNumbersFromFiles("data", "*")`
  - To get data from all files having "exam" in the name, `data = getNumbersFromFiles("data", "*exam*")`
  - To write this you need an accumulator to build a list of all the numbers.



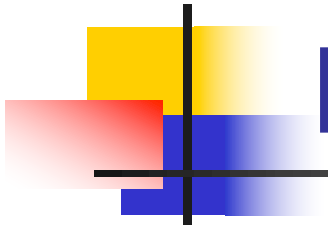
Iterating over Directories

```
def getNumbersFromFiles(basedir, pattern):
    path = Path(basedir)
    nums = []
    for filepath in path.glob(pattern):
        newnums = getNumbersFromFile(filepath)
        nums.extend(newnums)
    return nums
```



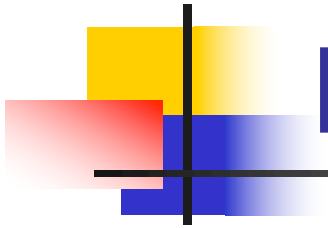
Iterating over Directories

- Notice how `basedir` was turned into a `Path` object at the start – that ensures that you can call `glob` in the heading.
- This function will work when `basedir` is passed as either a string or a `Path` object.



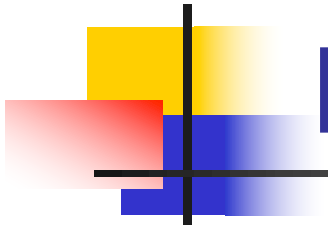
File Dialogs

- Some operating systems (e.g. Windows and macOS), by default will only show the main stem of the filename and not the type suffix, making it hard to know the full filename for performing file operations.
- This situation is even more complicated when the file exists somewhere other than the current working directory. In order to operate on these far-flung files, we need the complete path to them! Do you know how to find the complete path to an arbitrary file on your computer?



File Dialogs

- One solution to this problem is to allow users to browse the file system visually and navigate their way to particular file/directory.
- The usual technique incorporates a dialog box that allows a user to click around in the file system and either select or type in th ename of a file.
- Fortunately for us, the tkinter GUI library included with (most) standard Pythons has these kinds of functions!



File Dialogs

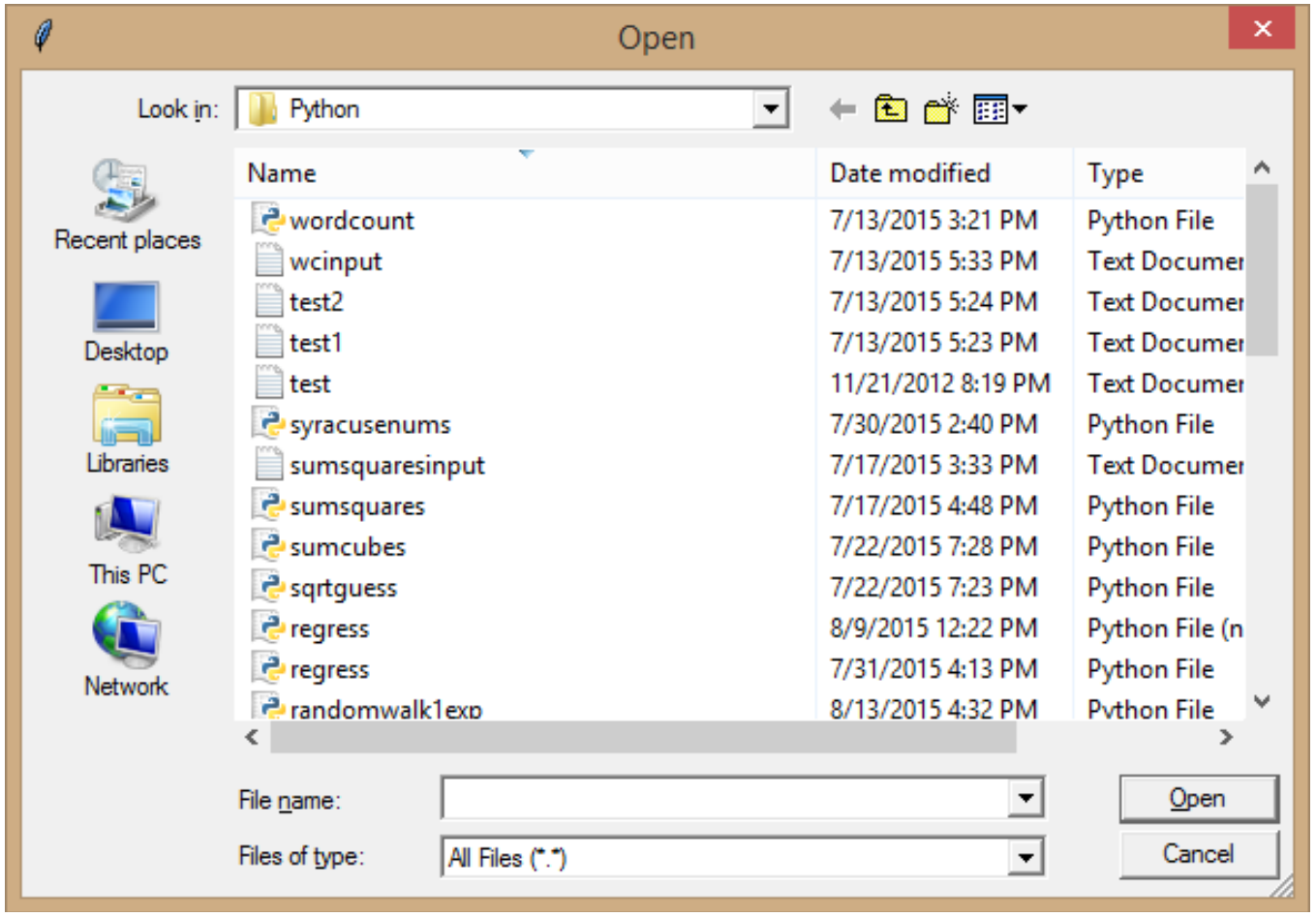
- To ask the user for the name of a file to open, you can use the `askopenfilename` function found in the `tkinter.filedialog` module.

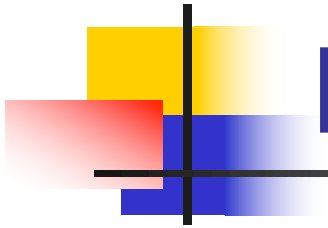
```
from tkinter.filedialog import askopenfilename
```

- The reason for the dot notation is that tkinter is package composed of multiple modules.
- To get the name of the user names file

```
infileName = askopenfilename()
```

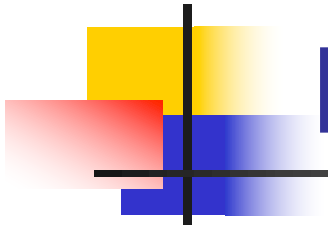
File Dialogs





File Dialogs

- The dialog box allows the user to either type in th ename of the file or to simply select it with the mouse.
- When the user clicks the “Open” button, the complete path name of the file is returned as a string and saved into the variable `infileName`.
- If the user clicks the “Cancel” button, the function will simpley return the empty string, "".



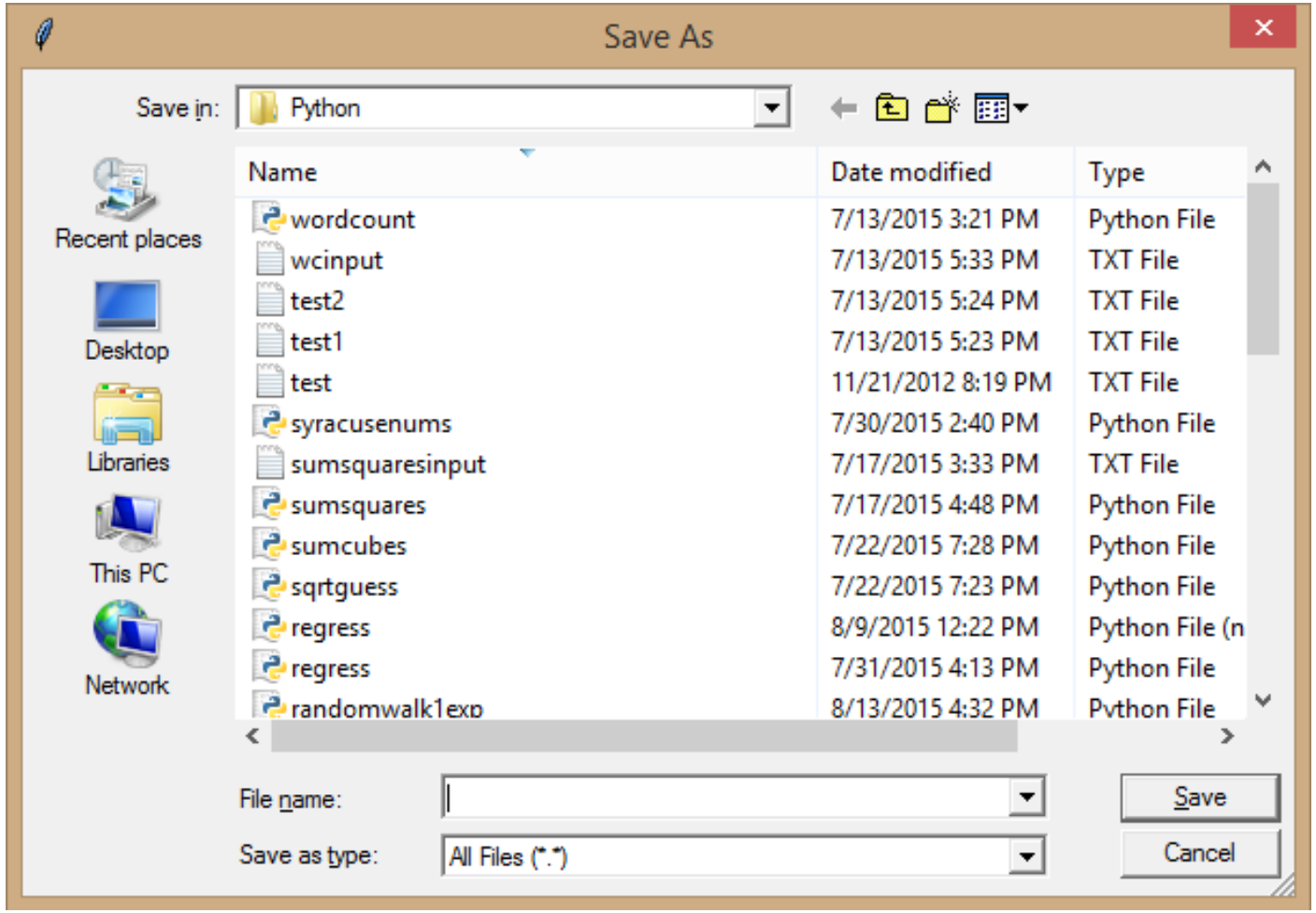
File Dialogs

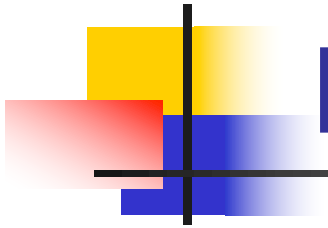
```
from tkinter.filedialog import asksaveasfilename
...
outfileName = asksaveasfilename()
```

- You could, of course, import both at once:

```
from tkinter.filedialog import askopenfilename, asksaveasfilename
```

File Dialogs





File Dialogs

- If you need to get a directory path from the user, there's also an `askdirectory` function.
- All these functions have numerous optional parameters that allow a program to customize the resulting dialogs.