

## Pattern Matching with Regular Expressions

*Regular expressions, called regexes for short, are descriptions for a pattern of text.*

### Finding Patterns of Text without Regular Expressions

Say you want to find a phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242. Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`. Open a new file editor window and enter the following code; then save the file as `isPhoneNumber.py`:

```
def isPhoneNumber(text):
    ❶ if len(text) != 12:
        return False
    for i in range(0, 3):
    ❷ if not text[i].isdecimal():
        return False
    ❸ if text[3] != '-':
        return False
    for i in range(4, 7):
    ❹ if not text[i].isdecimal():
        return False
    ❺ if text[7] != '-':
        return False
    for i in range(8, 12):
    ❻ if not text[i].isdecimal():
        return False
    ❼ return True

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

O/P

415-555-4242 is a phone number:

True

Moshi moshi is a phone number:

False

The `isPhoneNumber()` function has code that does several checks to see whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`.

You would have to add even more code to find this pattern of text in a larger string.

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
    ❶ chunk = message[i:i+12]
    ❷ if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

O/P

Phone number found: 415-555-1011

Phone number found: 415-555-9999

Done

On each iteration of the for loop, a new chunk of 12 characters from message is assigned to the variable chunk u. For example, on the first iteration, i is 0, and chunk is assigned message[0:12] (that is, the string 'Call me at 4'). On the next iteration, i is 1, and chunk is assigned message[1:13] (the string 'all me at 41').

You pass chunk to isPhoneNumber() to see whether it matches the phone number pattern v, and if so, you print the chunk. Continue to loop through message, and eventually the 12 characters in chunk will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies isPhoneNumber(). Once we're done going through message, we print Done.

### Finding Patterns of Text with Regular Expressions

The isPhoneNumber() function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99? The isPhoneNumber() function would fail to validate them. Y

For example, a \d in a regex stands for a digit character— that is, any single numeral 0 to 9. The regex \d\d\d-\d\d\d-\d\d\d-\d\d\d is used by Python to match the same text the previous isPhoneNumber() function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the \d\d\d-\d\d\d-\d\d\d-\d\d\d regex. But regular expressions can be much more sophisticated. For example, adding a 3 in curly brackets ({3}) after a pattern is like saying, “Match this pattern three times.” So the slightly shorter regex \d{3}-\d{3}-\d{4} also matches the correct phone number format.

## Creating Regex Objects

All the regex functions in Python are in the re module.

```
>>> import re
```

Passing a string value representing your regular expression to re.compile() returns a Regex pattern object (or simply, a Regex object).

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d')
```

However, by putting an r before the first quote of the string value, you can mark the string as a raw string, which does not escape characters. Since regular expressions frequently use backslashes in them, it is convenient to pass raw strings to the re.compile() function instead of typing extra backslashes. Typing r'\d\d\d-\d\d\d-\d\d\d' is much easier than typing '\\d\\d\\d-\\d\\d\\d-\\d\\d\\d'

## Matching Regex Objects

A Regex object's search () method searches the string it is passed for any matches to the regex. The search () method will return None if the regex pattern is not found in the string. If the pattern is found, the search () method returns a Match object. Match objects have a group () method that will return the actual matched text from the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d')
```

```
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
```

```
>>> print('Phone number found: ' + mo.group())
```

Phone number found: 415-555-4242

## Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's search() method. This returns a Match object.
4. Call the Match object's group() method to return a string of the actual matched text.

## More Pattern Matching with Regular Expressions

### Grouping with Parentheses

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create groups in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Then you can use the `group()` match object method to grab the matching text from just one group. The first set of parentheses in a regex string will be group 1. The second set will be group 2. By passing the integer 1 or 2 to the `group()` match object method, you can grab different parts of the matched text. Passing 0 or nothing to the `group()` method will return the entire matched text.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line

### Matching Multiple Groups with the Pipe

The `|` character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'. When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object.

```
>>> heroRegex = re.compile(r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses.

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

The method call `mo.group()` returns the full matched text 'Batmobile', while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, 'mobile'. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match. If you need to match an actual pipe character, escape it with a backslash, like `\\`.

### Regex-Quick Guide

- The `?` says the group matches zero or one times.
- The `*` says the group matches zero or more times.
- The `+` says the group matches one or more times.
- The curly braces can match a specific number of times.
- The curly braces with two numbers matches a minimum and maximum number of times.
- Leaving out the first or second number in the curly braces says there is no minimum or maximum.
- Greedy matching match the longest string possible, nongreedy matching match the shortest string possible.
- Putting a question mark after the curly braces makes it do a nongreedy match.

- `^` Matches the beginning of a line
- `$` Matches the end of the line
- `.` Matches any character
- `\s` Matches whitespace
- `\S` Matches any non-whitespace character
- `*` Repeats a character zero or more times
- `*?` Repeats a character zero or more times (non-greedy)
- `+` Repeats a character one or more times
- `+?` Repeats a character one or more times (non-greedy)

[aeiou] Matches a single character in the listed set

[^XYZ] Matches a single character not in the listed set

[a-z0-9] The set of characters can include a range

( Indicates where string extraction is to start

) Indicates where string extraction is to end

```
>>> import re
>>>
>>>
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo = batRegex.search('The Adventures of Batman')
>>> mo.group()
'Batman'
>>> mo = batRegex.search('The Adventures of Batwoman')
>>> mo.group()
'Batwoman'
>>> mo = batRegex.search('The Adventures of Batwowowowoman')
>>> mo == None
True
>>> phoneRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneRegex.search('My phone number is 415-555-1234. Call me tomorrow.')
>>> mo.group()
'415-555-1234'
>>> mo = phoneRegex.search('My phone number is 555-1234. Call me tomorrow.')
>>> mo == None
True
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> phoneRegex.search('My phone number is 415-555-1234. Call me tomorrow.')
<_sre.SRE_Match object; span=(19, 31), match='415-555-1234'>
>>> phoneRegex.search('My phone number is 555-1234. Call me tomorrow.')
<_sre.SRE_Match object; span=(19, 27), match='555-1234'>
```

### Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The ? Character flags the group that precedes it as an optional part of the pattern. If you need to match an actual question mark character, escape it with \?

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```



### Matching Zero or More with the Star(\*)

The \* (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let’s look at the Batman example again. If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

### Matching One or More with the Plus

While \* means “match zero or more,” the + (or plus) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear at least once. It is not optional. If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> batRegex.search('The Adventures of Batman')
<_sre.SRE_Match object; span=(18, 24), match='Batman'>
>>> batRegex.search('The Adventures of Batwoman')
<_sre.SRE_Match object; span=(18, 26), match='Batwoman'>
>>> batRegex.search('The Adventures of Batwowowowowowoman')
<_sre.SRE_Match object; span=(18, 38), match='Batwowowowowowoman'>
>>>
>>>
>>>
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> batRegex.search('The Adventures of Batman')
>>> batRegex.search('The Adventures of Batman') == None
True
>>> batRegex.search('The Adventures of Batwoman')
<_sre.SRE_Match object; span=(18, 26), match='Batwoman'>
>>> batRegex.search('The Adventures of Batwowowowowowoman')
<_sre.SRE_Match object; span=(18, 38), match='Batwowowowowowoman'>

>>> regex = re.compile(r'\+*\?')
>>> regex.search('I learned about +*? regex syntax')
<_sre.SRE_Match object; span=(16, 19), match='+*?'>
>>> regex = re.compile(r'(\+*\?)')
>>> regex.search('I learned about +*?*?*?*?*? regex syntax')
<_sre.SRE_Match object; span=(16, 31), match='+*?*?*?*?*?*?'>
```

### Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match `'HaHaHa'`, `'HaHaHaHa'`, and `'HaHaHaHaHa'`. You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

`(Ha){3}`

`(Ha)(Ha)(Ha)`

`(Ha){3,5}`

`((Ha)(Ha)(Ha))((Ha)(Ha)(Ha)(Ha))((Ha)(Ha)(Ha)(Ha)(Ha))`

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
```

```
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

### Greedy and Nongreedy Matching

Since `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the Pattern Matching with Regular Expressions 157 previous curly bracket example returns `'HaHaHaHaHa'` instead of the shorter possibilities. After all, `'HaHaHa'` and `'HaHaHaHa'` are also valid matches of the regular expression `(Ha){3,5}`.

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The nongreedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark. Enter the following into the interactive shell, and notice the difference between the greedy and nongreedy forms of the curly brackets searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```



## The findall() Method

In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the first matched text in the searched string, the findall() method will return the strings of every match in the searched string.

findall() will not return a Match object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there are groups in the regular expression, then findall() will return a list of tuples. Each tuple represents a found match, and its items are matched strings for each group in the regex. To see findall() in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[( '415', '555', '1122'), ('212', '555', '0000')]
```

To summarize what the findall() method returns, remember the following:

1. When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method findall() returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.
2. When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method findall() returns a list of tuples of strings (one string for each group), such as `[( '415', '555', '1122'), ('212', '555', '0000')]`.

**Character Classes-** `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`. There are many such shorthand character classes, as shown in Table below.

| Shorthand character class | Represents   |
|---------------------------|--|
| <code>\d</code>           | Any numeric digit from 0 to 9.   |
| <code>\D</code>           | Any character that is <i>not</i> a numeric digit from 0 to 9.  |
| <code>\w</code>           | Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.) |
| <code>\W</code>           | Any character that is <i>not</i> a letter, numeric digit, or the underscore character.                 |
| <code>\s</code>           | Any space, tab, or newline character. (Think of this as matching “space” characters.)                  |
| <code>\S</code>           | Any character that is <i>not</i> a space, tab, or newline.   |

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers 0 to 5; this is much shorter than typing `(0|1|2|3|4|5)`.

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `findall()` method returns all matching strings of the regex pattern in a list.

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

### Making Your Own Character Classes

You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase.

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash.

For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\.]`. By placing a caret character (`^`) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class.

```
>>> consonantRegex = re.compile(r'^[aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ',
',', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

### The Caret and Dollar Sign Characters

You can also use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the beginning of the searched text. Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate the string must end with this regex pattern. And you can use the `^` and `$` together to indicate that the entire string must match the regex.

**“Carrots cost dollars”** to remind that the **caret comes first and the dollar sign comes last**.

For example, the `r'^Hello'` regular expression string matches strings that begin with 'Hello'. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used.

## The Wildcard Character

The `.` (or dot) character in a regular expression is called a wildcard and will match any character except for a newline.

Remember that the dot character will match just one character, which is why the match for the text flat in the previous example matched only lat. To match an actual dot, escape the dot with a backslash: `\.`

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

### Matching Everything with Dot-Star

Sometimes you will want to match everything and anything. For example, say you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again. You can use the dot-star (.) to stand in for that “anything.” Remember that the dot character means “any single character except the newline,” and the star character means “zero or more of the preceding character.”

```
>>> nameRegex = re.compile(r'First Name: (.) Last Name: (.)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.?). Like with curly brackets, the question mark tells Python to match in a nongreedy way

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<to serve man> for dinner.>'
```

Both regexes roughly translate to “Match an opening angle bracket, followed by anything, followed by a closing angle bracket.” But the string ' for dinner.>' has two possible matches for the closing angle bracket. In the nongreedy version of the regex, Python matches the shortest possible string: ". In the greedy version, Python matches the longest possible string: ' for dinner.>'

**Matching Newlines with the Dot Character** The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match all characters, including the newline character.

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

### Review of Regex Symbols

- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly `n` of the preceding group.
- The `{n,}` matches `n` or more of the preceding group.
- The `{,m}` matches 0 to `m` of the preceding group.
- The `{n,m}` matches at least `n` and at most `m` of the preceding group.
- `{n,m}?` or `*?` or `+?` performs a nongreedy match of the preceding group.
- `^spam` means the string must begin with spam.
- `spam$` means the string must end with spam.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.
- `[abc]` matches any character between the brackets (such as `a`, `b`, or `c`).
- `[^abc]` matches any character that isn't between the brackets.

### Case-Insensitive Matching

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
```



```
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`.

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

### Substituting Strings with the sub() Method

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.” For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`. The `\1` in that string will be replaced by whatever text was matched by group 1— that is, the `(\w)` group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')
'A**** told C**** that E**** knew B**** was a double agent.'
```

### Managing Complex Regexes

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the `re.compile()` function to ignore whitespace and comments inside the regular

expression string. This “verbose mode” can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

---

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\\s|-|\\.)?\d{3}(\\s|-|\\.)\d{4}
(\\s*(ext|x|ext.)\\s*\d{2,5}))?')
```

---

you can spread the regular expression over multiple lines with comments like this:

---

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?      # area code
    (\\s|-|\\.)?          # separator
    \d{3}                  # first 3 digits
    (\\s|-|\\.)           # separator
    \d{4}                  # last 4 digits
    (\\s*(ext|x|ext.)\\s*\d{2,5})? # extension
    )''', re.VERBOSE)
```

---

### Combining `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE`

What if you want to use `re.VERBOSE` to write comments in your regular expression but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument. You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the bitwise or operator. Pattern Matching with Regular Expressions 165 So if you want a regular expression that’s case-insensitive and includes newlines to match the dot character, you would form your `re.compile()` call like this

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

### Syllabus: Regex

*Pattern Matching with Regular Expressions, Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The `findall()` Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the `sub()` Method, Managing Complex Regexes, Combining `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE`, Project: Phone Number and Email Address Extractor.*

## Files and os module

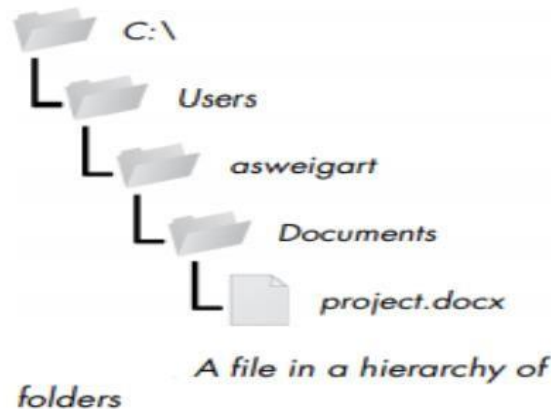
**How to use Python to create, read, and save files on the hard drive.**

### Syllabus

*Reading and Writing Files, Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module, Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard, Organizing Files, The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates, Project: Backing Up a Folder into a ZIP File, Debugging, Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger.*

### Files and File Paths

- A file has two key properties: a filename (usually written as one word) and a path.
- The path specifies the location of a file on the computer
- The C:\ part of the path is the root folder, which contains all other folders.
- On Windows, the root folder is named C:\ and is also called the C: drive.
- On Linux, the root folder is /.



Additional volumes, such as a DVD drive or USB thumb drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as D:\ or E:\. On OS X, they appear as new folders under the /Volumes folder. On Linux, they appear as new folders under the /mnt (“mount”) folder.

### Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases. Fortunately, this is simple to do with the os.path.join() function. If you pass it the string values of

individual file and folder names in your path, `os.path.join()` will return a string with a file path using the correct path separators.

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'

>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
    print(os.path.join('C:\\Users\\asweigart', filename))
C:\\Users\\asweigart\\accounts.txt
C:\\Users\\asweigart\\details.csv
C:\\Users\\asweigart\\invite.docx
```

### The Current Working Directory

Every program that runs on your computer has a current working directory, or `cwd`. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the `os.getcwd()` function and change it with `os.chdir()`.

```
>>> import os
>>> os.getcwd()
'C:\\Python34'

>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Python will display an error if you try to change to a directory that does not exist.

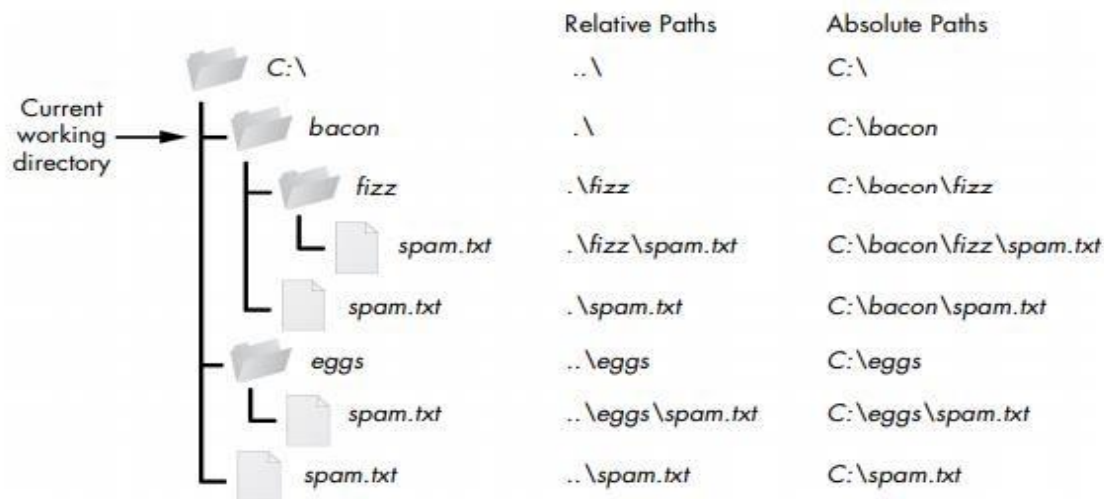
```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

## Absolute vs. Relative Paths

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's current working directory.

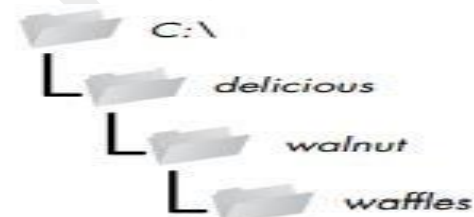
There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder.. When the current working directory is set to C:\bacon, the relative paths for the other folders and files are set as they are in the figure. The .\ at the start of a relative path is optional. For example, .\spam.txt and spam.txt refer to the same file.



*The relative paths for folders and files in the working directory C:\bacon*

## Creating New Folders with os.makedirs()

Your programs can create new folders (directories) with the os.makedirs() function. This will create not just the C:\delicious folder but also a walnut folder inside C:\delicious and a waffles folder inside C:\delicious\walnut. That is, os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists.



*The result of  
os.makedirs('C:\\delicious\\walnut\\waffles')*



```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

## The os.path Module

The os.path module contains many helpful functions related to filenames and file paths. For instance, you've already used os.path.join() to build paths in a way that will work on any operating system. Since os.path is a module inside the os module, you can import it by simply running import os. Whenever your programs need to work with files, folders, or file paths. The full documentation for the os.path module is on the Python website at <http://docs.python.org/3/library/os.path.html>.

## Handling Absolute and Relative Paths

The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

- Calling os.path.abspath(path) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one
- Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path.
- Calling os.path.relpath(path, start) will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path.

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

|                                 |           |
|---------------------------------|-----------|
| C:\\Windows\\System32\\calc.exe |           |
| Dir name                        | Base name |

*The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.*

Calling `os.path.dirname(path)` will return a string of everything that comes before the last slash in the path argument. Calling `os.path.basename(path)` will return a string of everything that comes after the last slash in the path argument.

### os module methods and values

- `os.name`
- `os.environ`
- `os.chdir()`
- `os.getcwd()`
- `os.getenv()`
- `os.putenv()`
- `os.mkdir()`
- `os.makedirs()`
- `os.remove()`
- `os.rename()`
- `os.rmdir()`
- `os.startfile()`
- `os.walk()`
- `os.path`

The `os.path` sub-module of the `os` module has lots of great functionality built into it.

- `basename`
- `dirname`
- `exists`
- `isdir` and `isfile`
- `join`
- `split`
- **`os.path.basename`**  
The `basename` function will return just the filename of a path.
- **`os.path.dirname`**  
The **`dirname`** function will return just the directory portion of the path.

```
path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```
- **`os.path.exists`**  
The **`exists`** function will tell you if a path exists or not
- **`os.path.isdir` / `os.path.isfile`**

The `isdir` and `isfile` methods are closely related to the `exists` method in that they also test for existence. However, `isdir` only checks if the path is a directory and `isfile` only checks if the path is a file.

### **os.path.join**

The `join` method give you the ability to join one or more path components together using the appropriate separator. For example, on Windows, the separator is the backslash, but on Linux, the separator is the forward slash.

```
>>> os.path.join(r'C:\Python27\Tools\pynche', 'hello.py')
'C:\\Python27\\Tools\\pynche\\hello.py'
```

### **os.path.split**

The **split** method will split a path into a tuple that contains the directory and the file.

```
>>> os.path.split(r'C:\Python27\Tools\pynche\hello.py')
('C:\\Python27\\Tools\\pynche', 'hello.py')
```

**what happens if the path doesn't have a filename on the end:**

```
>>> os.path.split(r'C:\Python27\Tools\pynche')
('C:\\Python27\\Tools', 'pynche')
```

### **Common use case of the split**

```
>>> dirname, fname = os.path.split(r'C:\Python27\Tools\pynche\hello.py')
```

```
>>> dirname
```

```
'C:\\Python27\\Tools\\pynche'
```

```
>>> fname
```

```
'hello.py'
```

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
```

```
>>> for filename in myFiles:
```

```
    print(os.path.join('C:\\Users\\asweigart',filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

```
>>> import os
```

```
>>> os.getcwd()
```

```
'C:\\Python34'
```

```
>>> os.chdir('C:\\Windows\\System32')
```

```
>>> os.getcwd()
```

```
'C:\\Windows\\System32'
```

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
```

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

```
os.chdir('C:\\ThisFolderDoesNotExist')
```

FileNotFoundError: [WinError 2] The system cannot find the file specified:

```
'C:\\ThisFolderDoesNotExist'
```

```
>>> import os
```

```
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

```
os.path.dirname(path)
```

```
os.path.basename(path)
```

*C:\\Windows\\System32\\calc.exe*

|                       |           |
|-----------------------|-----------|
| C:\\Windows\\System32 | calc.exe  |
| Dir name              | Base name |

*The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.*

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
```

```
>>> os.path.basename(path)
```

```
'calc.exe'
```

```
>>> os.path.dirname(path)
```

```
'C:\\Windows\\System32'
```

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
```

```
>>> os.path.split(calcFilePath)
```

```
('C:\\Windows\\System32', 'calc.exe')
```

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
```

```
('C:\\Windows\\System32', 'calc.exe')
```

```
>>> calcFilePath.split(os.path.sep)
```

```
['C:', 'Windows', 'System32', 'calc.exe']
```

- **Calling `os.path.getsize(path)` will return the size in bytes of the file in the path argument.**
- **Calling `os.listdir(path)` will return a list of filename strings for each file in the path argument.**

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
```

```
776192
```

```
>>> os.listdir('C:\\Windows\\System32')
```

```
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
```

```
--snip--
```

```
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

### **Find the total size of files in bytes in specified directory**

```
>>> totalSize = 0
```

```
>>> for filename in os.listdir('C:\\Windows\\System32'):
```

```
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
```

```
>>> print(totalSize)
```

```
1117846456
```

### **Checking Path Validity**

- Calling `os.path.exists(path)` will return True if the file or folder referred to in the argument exists and will return False if it does not exist.
- Calling `os.path.isfile(path)` will return True if the path argument exists and is a file and will return False otherwise.
- Calling `os.path.isdir(path)` will return True if the path argument exists and is a folder and will return False otherwise.

```
>>> os.path.exists('C:\\Windows')
```

```
True
```

```
>>> os.path.exists('C:\\some_made_up_folder')
```



False

```
>>> os.path.isdir('C:\\Windows\\System32')
```

True

```
>>> os.path.isfile('C:\\Windows\\System32')
```

False

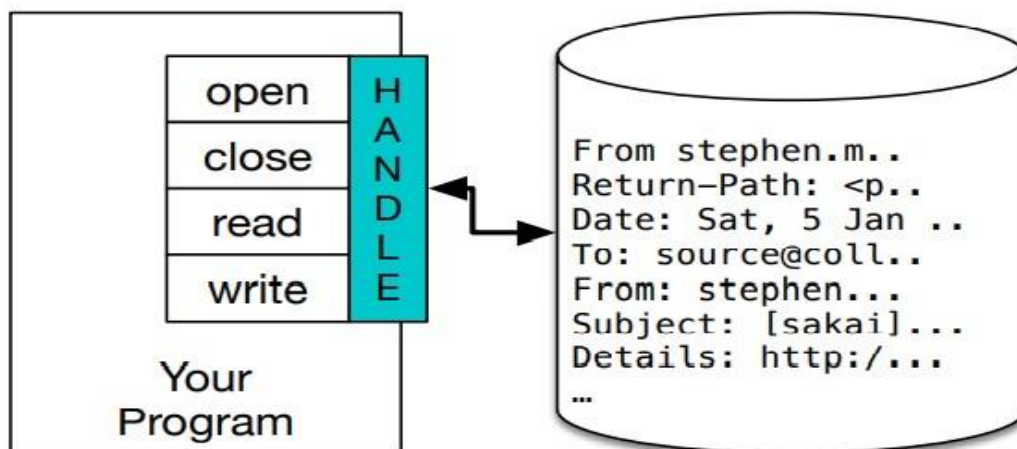
```
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
```

False

```
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
```

True

## File Handling-Python



### The File Reading/Writing Process

Before we do any interaction with file we must tell Python which file we are going to work.

There are three steps to reading or writing files in Python.

1. Call the `open()` function to return a File object.
2. Call the `read()` or `write()` method on the File object.
3. Close the file by calling the `close()` method on the File object.
  - To open a file with the `open()` function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path.
  - The `open()` function returns a File object.

## Opening Files with the open() Function

- To open a file with the open() function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path.
- The open() function returns a File object.

```
helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

## Reading the Contents of Files

If you want to read the entire contents of a file as a string value, use the File object's read() method.

```
>>> helloContent = helloFile.read()
```

```
>>> helloContent
```

```
'Hello world!'
```

**readlines() method to get a list of string values from the file, one string for each line of text**

```
>>> sonnetFile = open('sonnet29.txt')
```

```
>>> sonnetFile.readlines()
```

```
[When, in disgrace with fortune and men's eyes,\n' I all alone beweep my outcast state,\n' And trouble deaf heaven with my bootless cries,\n' And look upon myself and curse my fate,]
```

*Note that each of the string values ends with a newline character, \n , except for the last line of the file. A list of strings is often easier to work with than a single large string value.*

## Writing to Files

- You can't write to a file you've opened in read mode, though.
- Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, or write mode and append mode for short.
- Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value.
- Pass 'w' as the second argument to open() to open the file in write mode.
- Append mode, on the other hand, will append text to the end of the existing file.
- You can think of this as appending to a list in a variable, rather than overwriting the variable altogether.
- Pass 'a' as the second argument to open() to open the file in append mode.
- If the filename passed to open() does not exist, both write and append mode will create a new, blank file.
- After reading or writing a file, call the close() method before opening the file again.

```
>>> fhand = open('mbox.txt')
```

```
>>> print(fhand)
```

```
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

- Calling write() on the opened file and passing write() the string argument 'Hello world! /n' writes the string to the file and returns the number of characters written, including the newline. Then we close the file.
- To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write 'Bacon is not a vegetable.' to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call read(), store the resulting File object in content, close the file, and print content

**Program to count and display number of lines in the given file.**

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)
>>> fhand = open('mbox-short.txt')
```

```
>>> inp = fhand.read()
```

```
>>> print(len(inp))
```

```
94626
```

```
>>> print(inp[:20])
```

```
From stephen.marquar
```

**Program to count and display number of lines starts by *From* in the given file.**

```
fhand = open('mbox-short.txt')
```

```
count = 0
```

```
for line in fhand:
```

```
    if line.startswith('From:')
```

```
        print(line)
```

### **Saving Variables with the shelf Module**

The shelf module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

#### **How to store program data to file?**

```
>>> import shelve
```

```
>>> shelfFile = shelve.open('mydata')
```

```
>>> cats = ['Zophie', 'Pooka', 'Simon']
```

```
>>> shelfFile['cats'] = cats
```

```
>>> shelfFile.close()
```

- After running the previous code on Windows, you will see three new files in the current working directory: mydata.bak, mydata.dat, and mydata.dir
- programs can use the shelf module to later reopen and retrieve the data from these shelf files.
- Shelf values don't have to be opened in read or write mode—they can do both once opened.

```
>>> shelfFile = shelve.open('mydata')
```

```
>>> type(shelfFile)
```

```
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

### **Saving Variables with the pprint.pformat() Function**

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
>>> fileObj.close()
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```



## File Organization-The shutil Module

- The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs.
- To use the shutil functions, you will first need to use import shutil.
- The shutil module provides functions for copying files, as well as entire folders.
- Calling `shutil.copy(source, destination)` will copy the file at the path source to the folder at the path destination. (Both source and destination are strings.)
- If destination is a filename, it will be used as the new name of the copied file.
- This function returns a string of the path of the copied file

### shutil.copy()

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

### shutil.copytree()

- While `shutil.copy()` will copy a single file.
- The `shutil.copytree()` copy an entire folder and every folder and file contained in it.
- Calling `shutil.copytree(source, destination)` will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination.
- The source and destination parameters are both strings.
- The function returns a string of the path of the copied folder.

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

## Moving and Renaming Files and Folders

- Calling **`shutil.move(source, destination)`** will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location.
- If destination points to a folder, the source file gets moved into destination and keeps its current filename.

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named eggs already exists in the C:\ directory, this shutil.move() calls says, “Move C:\bacon.txt into the folder C:\eggs.” If there had been a bacon.txt file already in C:\eggs, it would have been overwritten. Since it’s easy to accidentally overwrite files in this way, you should take some care when using move().

**The destination path can also specify a filename. In the following example, the source file is moved and renamed.**

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

- This line says, “Move C:\bacon.txt into the folder C:\eggs, and while you’re at it, rename that bacon.txt file to new\_bacon.txt.”
- Both of the previous examples worked under the assumption that there was a folder eggs in the C:\ directory. But if there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell.

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
  File "C:\Python34\lib\shutil.py", line 533, in move
    copy2(src, real_dst)
  File "C:\Python34\lib\shutil.py", line 244, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
  File "C:\Python34\lib\shutil.py", line 108, in copyfile
    with open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\
eggs\\ham'
```

## Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module.

- Calling `os.unlink(path)` will delete the file at path.
- Calling `os.rmdir(path)` will delete the folder at path.

This folder must be empty of any files or folders.

- Calling `shutil.rmtree(path)` will remove the folder at path, and all files and folders it contains will also be deleted.

*Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and with `print()` calls added to show the files that would be deleted.*

```
import os
```

```
for filename in os.listdir():
```

```
    if filename.endswith('.txt'):
```

```
        os.unlink(filename)
```

```
import os
```

```
for filename in os.listdir():
```

```
    if filename.endswith('.rxt'):
```

```
        #os.unlink(filename)
```

```
        print(filename)
```

- `os.unlink()` call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted.
- Once you are certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink (filename)` line. Then run the program again to actually delete the files.

## Safe Deletes with the `send2trash` Module

Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party `send2trash` module. You can install this module by running `pip install send2trash` from a Terminal window.

```
>>> import send2trash
```

```
>>> baconFile = open('bacon.txt', 'a') # creates the file
```

```
>>> baconFile.write('Bacon is not a vegetable.')
```

```
25
```

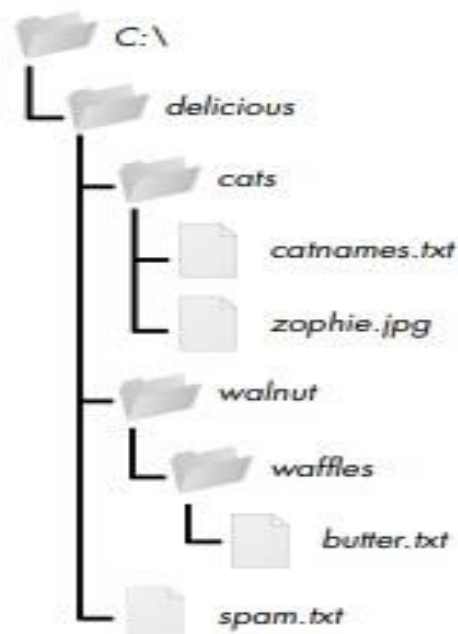
```
>>> baconFile.close()
```

```
>>> send2trash.send2trash('bacon.txt')
```

- In general, you should always use the `send2trash.send2trash()` function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does.
- If you want your program to free up disk space, use the `os` and `shutil` functions for deleting files and folders. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

## Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky fortunately, Python provides a function to handle this process for you.



*An example folder that  
contains three folders and four files*

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)

    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': ' + filename)

    print('')
```

The `os.walk()` function is passed a single string value: the path of a folder.

You can use `os.walk()` in a for loop statement to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers.

Unlike `range ()`, the `os.walk ()` function will return three values on each iteration through the loop:

1. A string of the current folder's name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

Just like you can choose the variable name `i` in the code for `i in range (10)`, you can also choose the variable names for the three values listed earlier. I usually use the names `folder name`, `subfolders`, and `filenames`. When you run this program, it will output the following.

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since `os.walk()` returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops. Replace the `print()` function calls with your own custom code.

## Compressing Files with the zipfile Module

- You may be familiar with ZIP files (with the `.zip` file extension), which can hold the compressed contents of many other files.
- Compressing a file reduces its size, which is useful when transferring it over the Internet. Since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an archive file.
- Your Python programs can both create and open (or extract) ZIP files using functions in the `zipfile` module.



## Reading ZIP Files

- To read the contents of a ZIP file, first you must create a `ZipFile` object.
- `ZipFile` objects are conceptually similar to the `File` objects.
- They are values through which the program interacts with the file.
- To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the `.zip` file's filename.
- Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.



```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

- A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file.
- These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file.
- ZipInfo objects have their own attributes, such as file\_size and compress\_size in bytes, which hold integers of the original file size and compressed file size, respectively.
- While a ZipFile object represents an entire archive file, a ZipInfo object holds useful information about a single file in the archive.

The command number 1. calculates how efficiently example.zip is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with %s.

## Extracting from ZIP Files

The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of example.zip will be extracted to C:\. Optionally, you can pass a folder name to extractall() to have it extract the files into a folder other than the current working directory. If the folder passed to the extractall() method does not exist, it will be created.

**The extract() method for ZipFile objects will extract a single file from the ZIP file**

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

- The string you pass to extract() must match one of the strings in the list returned by namelist(). Optionally, you can pass a second argument to extract() to extract the file into a folder other than the current working directory.
- If this second argument is a folder that doesn't yet exist, Python will create the folder.
- The value that extract() returns is the absolute path to which the file was extracted.

## Creating and Adding to ZIP Files

- To create your own compressed ZIP files, you must open the ZipFile object in write mode by passing 'w' as the second argument. (This is similar to opening a text file in write mode by passing 'w' to the open() function.)
- When you pass a path to the write() method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file.
- The write() method's first argument is a string of the filename to add.
- The second argument is the compression type parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to zipfile.
- ZIP\_DEFLATED. (This specifies the deflate compression algorithm, which works well on all types of data.)

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

The above code will create a new ZIP file named new.zip that has the compressed contents of spam.txt.

**Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in append mode**

## DEBUGGING

To paraphrase an old joke among programmers, writing code accounts for 90 percent of programming. Debugging code accounts for the other 90 percent. Your computer will do only what you tell it to do; it won't read your mind and do what you intended it to do. Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem. Fortunately, there are a few tools and techniques to identify what exactly your code is doing and where it's going wrong.

- First, you will look at logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.
- Second, you will look at how to use the debugger. The debugger is a feature that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track how the values change over the course of your program. This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be from the source code.

## RAISING EXCEPTIONS

Python raises an exception whenever it tries to execute invalid code. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the except statement. "Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function

**For example, enter the following into the interactive shell:**

```
>>> raise Exception("This is the error message.")
Traceback (most recent call last):
File "<pyshell#191>", line 1, in <module>
raise Exception("This is the error message.")
Exception: This is the error message.
```

If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception's error message. Often it's the code that calls the function, rather than the function itself, that knows how to handle an exception. That means you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

For example, open a new file editor tab, enter the following code, and save the program as boxPrint.py:

```
def boxPrint(symbol, width, height):
```

```

if len(symbol) != 1:
    raise Exception('Symbol must be a single character string.')
if width <= 2:
    raise Exception('Width must be greater than 2.')
if height <= 2:
    raise Exception('Height must be greater than 2.')
print(symbol * width)
for i in range(height - 2):
    print(symbol + (' ' * (width - 2)) + symbol)
print(symbol * width)
for sym, w, h in ((' ', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
    except Exception as err:
        print('An exception happened: ' + str(err))

```

we've defined a boxPrint() function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the screen. Say we want the character to be a single character, and the width and height to be greater than 2. We add if statements to raise exceptions if these requirements aren't satisfied. Later, when we call boxPrint() with various arguments, our try/except will handle invalid arguments.

This program uses the except Exception as err form of the except statement. If an Exception object is returned from boxPrint() this except statement will store it in a variable named err. We can then convert the Exception object to a string by passing it to str() to produce a user-friendly error message. When you run this boxPrint.py, the output will look like this:

```

****
* *
* *
****
00000000000000000000
0                      0
0                      0
0                      0
00000000000000000000
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.

```

An exception happened: Width must be greater than 2.

An exception happened: Symbol must be a single character string.

Using the try and except statements, you can handle errors more gracefully instead of letting the entire program crash.

## GETTING THE TRACEBACK AS A STRING

When Python encounters an error, it produces a treasure trove of error information called the traceback. The **traceback** includes the error message, the line number of the line that caused the

error, and the sequence of the function calls that led to the error. This sequence of calls is called the **call stack**.

Open a new file editor tab & enter the following program

```
def spam():  
    bacon()  
  
def bacon():  
    raise Exception("This is the error message.")  
  
spam()
```

**When you run the output will look like this:**

```
Traceback (most recent call last):  
File "errorExample.py", line 7, in <module>  
    spam()  
File "errorExample.py", line 2, in spam  
    bacon()  
File "errorExample.py", line 5, in bacon  
    raise Exception("This is the error message.")  
Exception: This is the error message.
```

From the traceback, you can see that the error happened on line 5, in the `bacon()` function. This particular call to `bacon()` came from line 2, in the `spam()` function, which in turn was called on line 7. In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

Python displays the traceback whenever a raised exception goes unhandled. But you can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a text file and keep your program running. You can look at the text file later, when you're ready to debug your program.

```
>>> import traceback  
  
>>> try:  
    raise Exception("This is the error message.")  
except:
```

```
errorFile = open('errorInfo.txt', 'w')
errorFile.write(traceback.format_exc())
errorFile.close()
```

**o/p**

**111**

**The traceback info was written to errorInfo.txt.**

The 111 is the return value from the write() method, since 111 characters were written to the file.

**The traceback text was written to errorInfo.txt and its content**

Traceback (most recent call last):

File "<pyshell#28>", line 2, in <module>

Exception: This is the error message.

## ASSERTIONS

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

- The assert keyword A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Here we've set podBayDoorStatus to 'open', so from now on, we fully expect the value of this variable to be 'open'. In a program that uses this variable, we might have written a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work as we expect. **So we add an assertion to make sure we're right to assume podBayDoorStatus is**



'open'. Here, we include the message 'The pod bay doors need to be "open".' so it'll be easy to see what's wrong if the assertion fails.

Later, say we make the obvious mistake of assigning podBayDoorStatus another value, but don't notice it among many lines of code. **The assertion catches this mistake and clearly tells us what's wrong.**

In plain English, an assert statement says, "I assert that the condition holds true, and if not, there is a bug somewhere, so immediately stop the program." For example, enter the following into the interactive shell:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.sort()
>>> ages
[15, 17, 22, 26, 47, 54, 57, 73, 80, 92]
>>> assert
ages[0] <= ages[-1] # Assert that the first age is <= the last age.
```

The assert statement here asserts that the first item in ages should be less than or equal to the last one. This is a sanity check; if the code in sort() is bug-free and did its job, then the assertion would be true.

Because the ages[0] <= ages[-1] expression evaluates to True, the assert statement does nothing.

However, let's pretend we had a bug in our code. Say we accidentally called the reverse() list method instead of the sort() list method. When we enter the following in the interactive shell, the assert statement raises an AssertionError:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.reverse()
>>> ages
[73, 47, 80, 17, 15, 22, 54, 92, 57, 26]
>>> assert ages[0] <= ages[-1] # Assert that the first age is <= the last age.
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

### Assertion Error

Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash. By "failing fast" like this, you shorten the time between the

original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the bug's cause.

Assertions are for programmer errors, not user errors. Assertions should only fail while the program is under development; a user should never see an assertion error in a finished program. For errors that your program can run into as a normal part of its operation (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement. You shouldn't use assert statements in place of raising exceptions, because users can choose to turn off assertions.

## Disabling Assertions

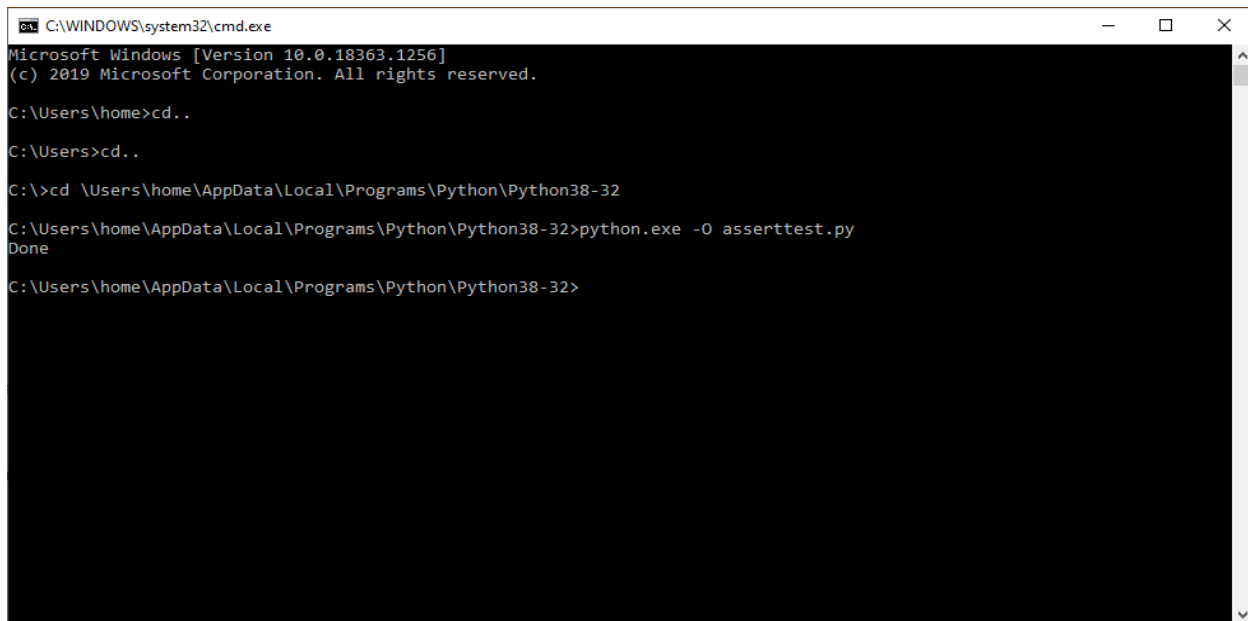
If you run a Python script with `python -O myscript.py` instead of `python myscript.py`, Python will skip assert statements. Users might disable assertions when they're developing a program and need to run it in a production setting that requires peak performance.

- Assertions can be disabled by passing the `-O` option when running Python.
- This is good for when you have finished writing and testing your program and don't want it to be slowed down by performing sanity checks (although
- Most of the time assert statements do not cause a noticeable speed difference).
- Assertions are for development, not the final product.
- By the time you hand off your program to someone else to run, it should be free of bugs

asserttest.py - C:/Users/home/AppData/Local/Programs/Python/Python38-32/asserttest.py (3.8.1)  
File Edit Format Run Options Window Help

```
assert(False)  
print('Done')|
```

```
= RESTART: C:/Users/home/AppData/Local/Programs/Python/Python38-32/asserttest.py  
Traceback (most recent call last):  
  File "C:/Users/home/AppData/Local/Programs/Python/Python38-32/asserttest.py",  
    line 1, in <module>  
      assert(False)  
AssertionError
```



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\home>cd..
C:\Users>cd..
C:\>cd \Users\home\AppData\Local\Programs\Python\Python38-32
C:\Users\home\AppData\Local\Programs\Python\Python38-32>python.exe -O asserttest.py
Done
C:\Users\home\AppData\Local\Programs\Python\Python38-32>
```

**The o/p shows only outcome of print statement instead of assertion which was disabled.**

Assertions also aren't a replacement for comprehensive testing.

For instance, if the previous ages example was set to [10, 3, 2, 1, 20], then the `assert ages[0] <= ages[-1]` assertion wouldn't notice that the list was unsorted, because it just happened to have a first age that was less than or equal to the last age, which is the only thing the assertion checked for.

### Using an Assertion in a Traffic Light Simulation

Building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
```

```
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street. To start the project, you want to write a `switchLights()` function, which will take an intersection dictionary as an argument and switch the lights.

At first, you might think that `switchLights()` should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'. The code to implement this idea might look like this:

```
def switchLights(stoplight):  
    for key in stoplight.keys():  
        if stoplight[key] == 'green':  
            stoplight[key] = 'yellow'  
        elif stoplight[key] == 'yellow':  
            stoplight[key] = 'red'  
        elif stoplight[key] == 'red':  
            stoplight[key] = 'green'  
    switchLights(market_2nd)
```

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the `switchLights()` function. But if while writing `switchLights()` you had added an assertion to check that at least one of the lights is always red, you might have included the following at the bottom of the function:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

With this assertion in place, your program would crash with this error message:

Traceback (most recent call last):

File "carSim.py", line 14, in <module>

`switchLights(market_2nd)`

File "carSim.py", line 13, in `switchLights`

`assert 'red' in stoplight.values(), 'Neither light is red! ' +`

`str(stoplight)`

❶ `AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}`

The important line here is the `AssertionError` ❶. While your program crashing is not ideal, it immediately points out that a sanity check failed: neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program's execution, you can save yourself a lot of future debugging effort.

## LOGGING

If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of logging to debug your code. Logging is a great way to understand what's happening in your program and in what order it's happening. Python's logging module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

### Using the logging Module

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program

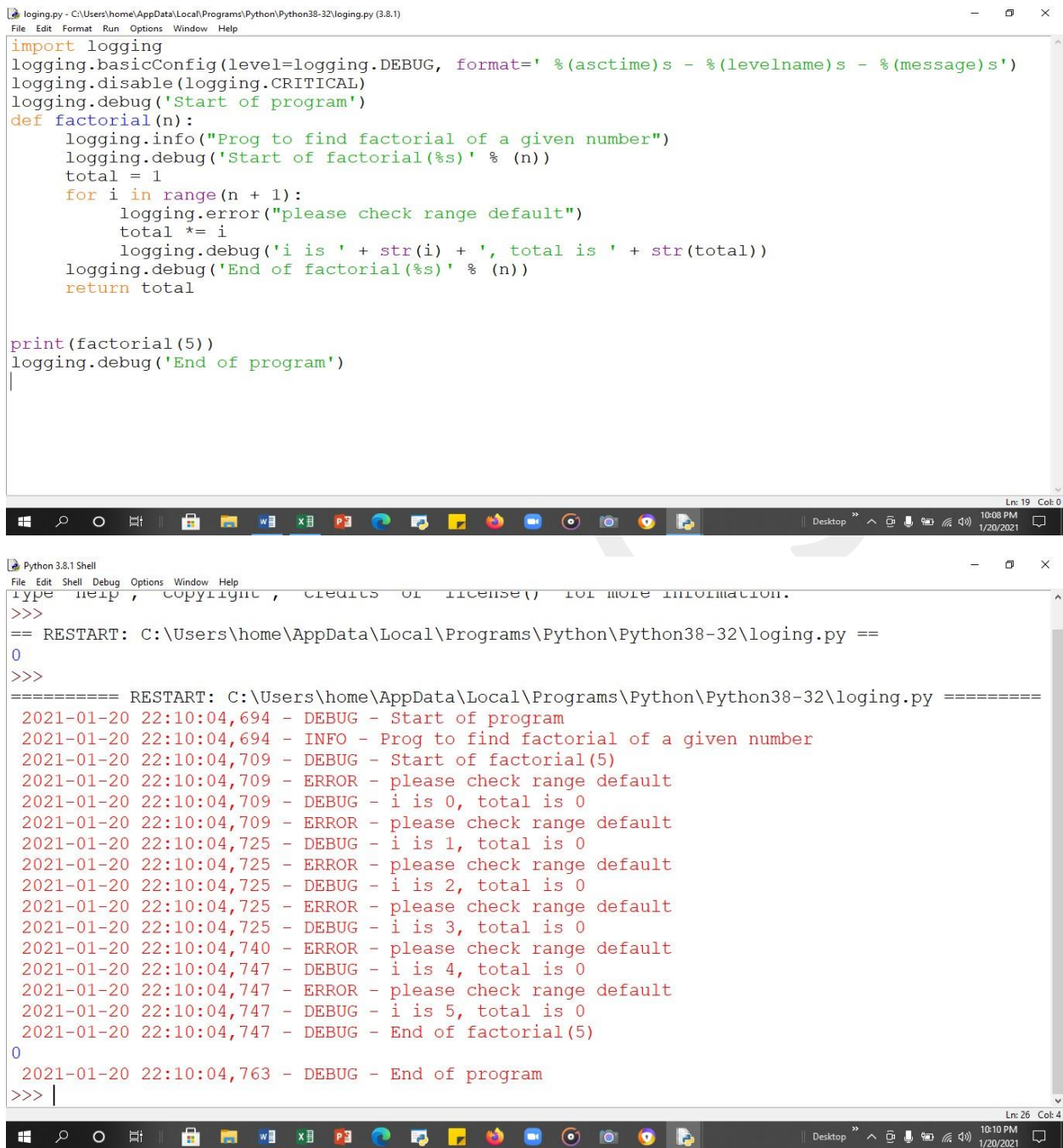
```
import logging
```

```
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

You don't need to worry too much about how this works, but basically, when Python logs an event, it creates a `LogRecord` object that holds information about that event. The logging module's `basicConfig()` function lets you specify what details about the `LogRecord` object you want to see and how you want those details displayed.

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is  $1 \times 2 \times 3 \times 4$ , or 24. Factorial 7 is  $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$ , or 5,040.

Open a new file editor tab and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as `factorialLog.py`.



```

logging.py - C:\Users\home\AppData\Local\Programs\Python\Python38-32\logging.py (3.8.1)
File Edit Format Run Options Window Help

import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logging.disable(logging.CRITICAL)
logging.debug('Start of program')
def factorial(n):
    logging.info("Prog to find factorial of a given number")
    logging.debug('Start of factorial(%)' % (n))
    total = 1
    for i in range(n + 1):
        logging.error("please check range default")
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%)' % (n))
    return total

print(factorial(5))
logging.debug('End of program')
|

Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
type help, copyright, credits or license() for more information.
>>>
== RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\logging.py ==
0
>>>
===== RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\logging.py =====
2021-01-20 22:10:04,694 - DEBUG - Start of program
2021-01-20 22:10:04,694 - INFO - Prog to find factorial of a given number
2021-01-20 22:10:04,709 - DEBUG - Start of factorial(5)
2021-01-20 22:10:04,709 - ERROR - please check range default
2021-01-20 22:10:04,709 - DEBUG - i is 0, total is 0
2021-01-20 22:10:04,709 - ERROR - please check range default
2021-01-20 22:10:04,725 - DEBUG - i is 1, total is 0
2021-01-20 22:10:04,725 - ERROR - please check range default
2021-01-20 22:10:04,725 - DEBUG - i is 2, total is 0
2021-01-20 22:10:04,725 - ERROR - please check range default
2021-01-20 22:10:04,725 - DEBUG - i is 3, total is 0
2021-01-20 22:10:04,740 - ERROR - please check range default
2021-01-20 22:10:04,747 - DEBUG - i is 4, total is 0
2021-01-20 22:10:04,747 - ERROR - please check range default
2021-01-20 22:10:04,747 - DEBUG - i is 5, total is 0
2021-01-20 22:10:04,747 - DEBUG - End of factorial(5)
0
2021-01-20 22:10:04,763 - DEBUG - End of program
>>> |

```

The factorial() function is returning 0 as the factorial of 5, which isn't right. The for loop should be multiplying the value in total by the numbers from 1 to 5. But the log messages displayed by logging.debug() show that the i variable is starting at 0 instead of 1. Since zero times anything is zero, the rest of the iterations also have the wrong value for total. Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.

**Change the for i in range(n + 1): line to for i in range(1, n + 1):, and run the program again.**

```
import logging

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

logging.disable(logging.CRITICAL)

logging.debug('Start of program')

def factorial(n):

    logging.info("Prog to find factorial of a given number")

    logging.debug('Start of factorial(%s)' % (n))

    total = 1

    #modification in range

    for i in range(1,n + 1):

        logging.error("please check range default")

        total *= i

        logging.debug('i is ' + str(i) + ', total is ' + str(total))

    logging.debug('End of factorial(%s)' % (n))

    return total

print(factorial(5))
```

logging.debug('End of program') Here, we use the logging.debug() function when we want to print log information. This debug() function will call basicConfig(), and a line of information will be printed. This information will be in the format we specified in basicConfig() and will include the messages we passed to debug(). The print(factorial(5)) call is part of the original program, so the result is displayed even if logging messages are disabled.



```

logging.py - C:\Users\home\AppData\Local\Programs\Python\Python38-32\logging.py (3.8.1)
File Edit Format Run Options Window Help
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
#logging.disable(logging.CRITICAL)
logging.debug('Start of program')
def factorial(n):
    logging.info("Prog to find factorial of a given number")
    logging.debug('Start of factorial(%s)' % (n))
    total = 1
    for i in range(1,n + 1):
        logging.error("please check range default")
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s)' % (n))
    return total

print(factorial(5))
logging.debug('End of program')

```

```

Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\logging.py ==
2021-01-20 22:04:58,320 - DEBUG - Start of program
2021-01-20 22:04:58,335 - INFO - Prog to find factorial of a given number
2021-01-20 22:04:58,335 - DEBUG - Start of factorial(5)
2021-01-20 22:04:58,335 - ERROR - please check range default
2021-01-20 22:04:58,335 - DEBUG - i is 1, total is 1
2021-01-20 22:04:58,335 - ERROR - please check range default
2021-01-20 22:04:58,335 - DEBUG - i is 2, total is 2
2021-01-20 22:04:58,351 - ERROR - please check range default
2021-01-20 22:04:58,351 - DEBUG - i is 3, total is 6
2021-01-20 22:04:58,351 - ERROR - please check range default
2021-01-20 22:04:58,351 - DEBUG - i is 4, total is 24
2021-01-20 22:04:58,367 - ERROR - please check range default
2021-01-20 22:04:58,367 - DEBUG - i is 5, total is 120
2021-01-20 22:04:58,367 - DEBUG - End of factorial(5)
120
2021-01-20 22:04:58,367 - DEBUG - End of program
>>>

```

The factorial(5) call correctly returns 120. The log messages showed what was going on inside the loop, which led straight to the bug. You can see that the logging.debug() calls printed out not just the strings passed to them but also a timestamp and the word DEBUG. Don't Debug with the print() Function.

Typing `import logging` and `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` is somewhat unwieldy.

You may want to use `print()` calls instead, but don't give in to this temptation! Once you're done debugging, you'll end up spending a lot of time removing `print()` calls from your code for each log message. You might even accidentally remove some `print()` calls that were being used for nonlog messages. The nice thing about log messages is that you're free to fill your program with as many as you like, and you can always disable them later by adding a single **`logging.disable(logging.CRITICAL)`** call. Unlike `print()`, the logging module makes it easy to switch between showing and hiding log messages.

Log messages are intended for the programmer, not the user. The user won't care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For messages that the user will want to see, like File not found or Invalid input, please enter a number, you should use a `print()` call. You don't want to deprive the user of useful information after you've disabled log messages.

### Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10.1 from least to most important. Messages can be logged at each level using a different logging function. Your logging message is passed as a string to these functions. The logging levels are

Table 10.1: Logging Levels in Python

| Level   | Logging Function               | Description  |
|---------|--------------------------------|--|
| DEBUG   | <code>logging.debug()</code>   | The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.                 |
| INFO    | <code>logging.info()</code>    | Used to record information on general events in your program or confirm that things are working at their point in the program. |
| WARNING | <code>logging.warning()</code> | Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.              |

| Level    | Logging Function   | Description  |
|----------|--------------------|--|
| ERROR    | logging.error()    | Used to record an error that caused the program to fail to do something.   |
| CRITICAL | logging.critical() | The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely. |

The benefit of logging levels is that you can change what priority of logging message you want to see. Passing logging.DEBUG to the basicConfig() function's level keyword argument will show messages from all the logging levels (DEBUG being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set basicConfig()'s level argument to logging.ERROR. This will show only ERROR and CRITICAL messages and skip the DEBUG, INFO, and WARNING messages.

```
>>> import logging
```

```
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

```
>>> logging.debug('Some debugging details.')
```

```
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
```

```
>>> logging.info('The logging module is working.')
```

```
2015-05-18 19:04:35,569 - INFO - The logging module is working.
```

```
>>> logging.warning('An error message is about to be logged.')
```

```
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
```

```
>>> logging.error('An error has occurred.')
```

```
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
```

```
>>> logging.critical('The program is unable to recover!')
```

```
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

## Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand. You simply pass `logging.disable()` a logging level, and it will suppress all log messages at that level or lower. So if you want to disable logging entirely, just add `logging.disable(logging.CRITICAL)` to your program. For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s - %(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2019-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since `logging.disable()` will disable all messages after it, you will probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

## Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

The log messages will be saved to `myProgramLog.txt`. While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program. You can open this text file in any text editor, such as Notepad or TextEdit.

## DEBUGGER

The debugger is a feature of the IDLE, and other editor software that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program “under the debugger” like this, you can take as much

time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.

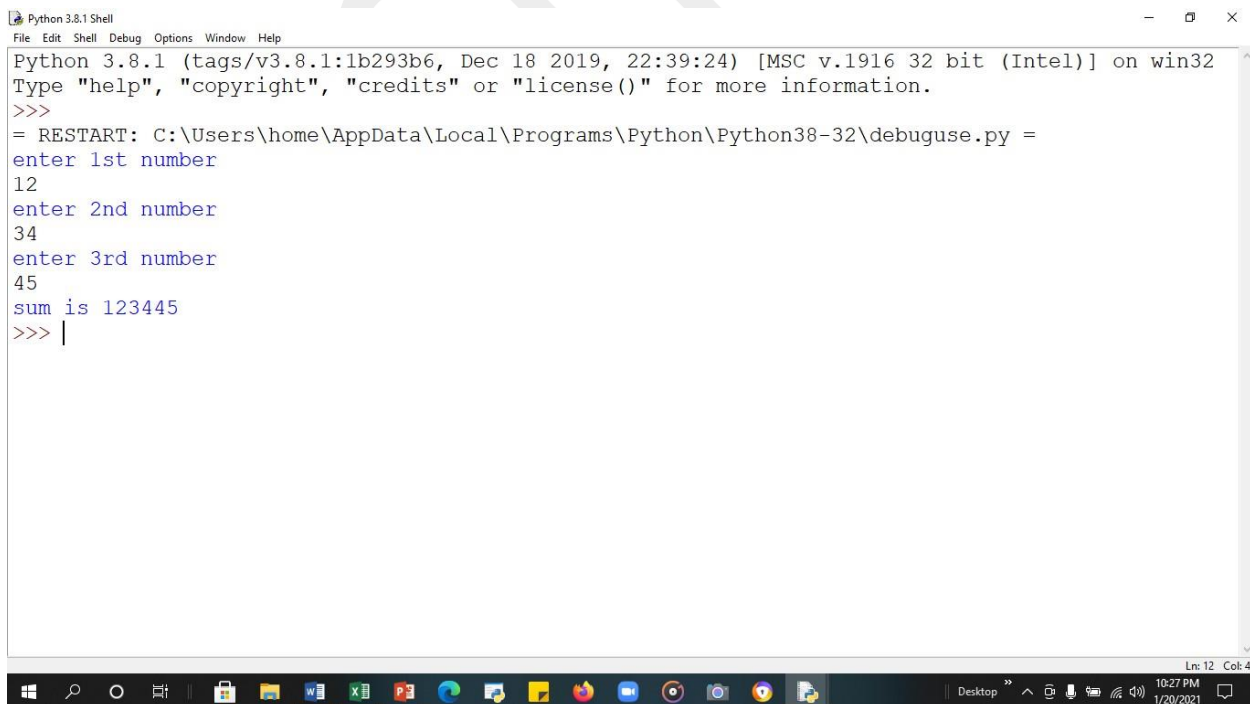
To run a program under debugger, click the Debug button in the top row of buttons, next to the Run button. Along with the usual output pane at the bottom, the Debug Inspector pane will open along the right side of the window.

### Debugging a Number Adding Program

#### Debugus.py

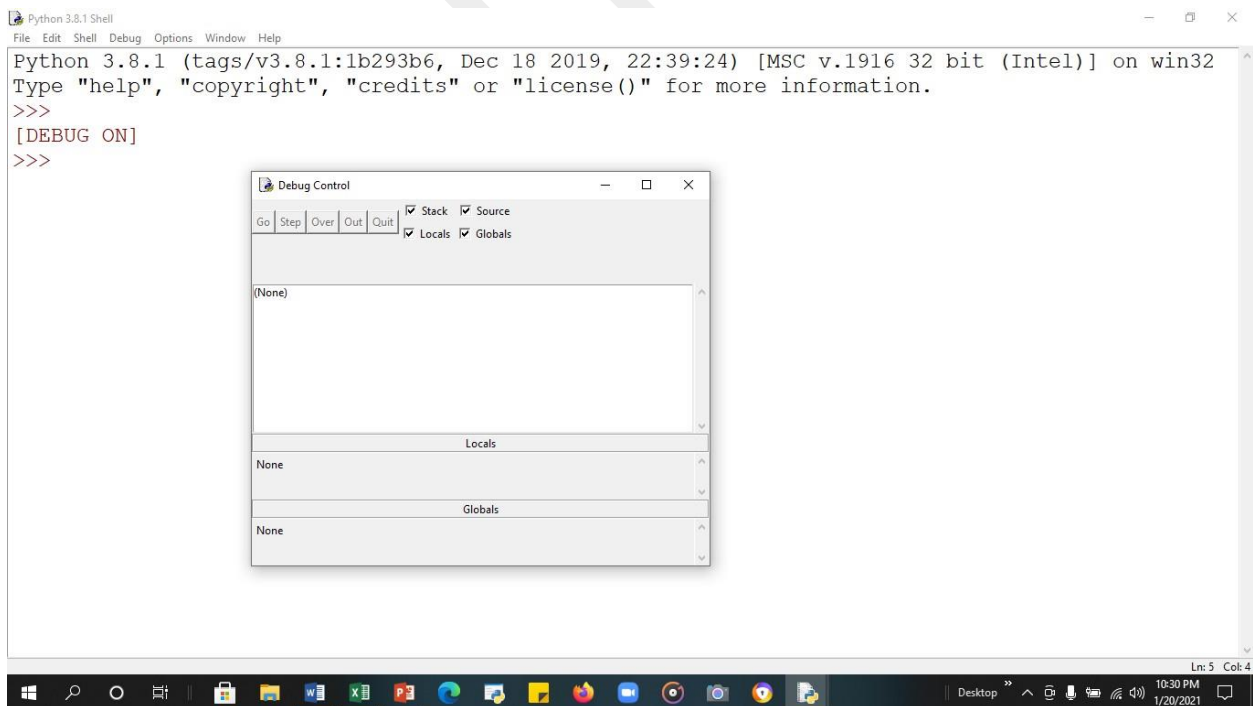
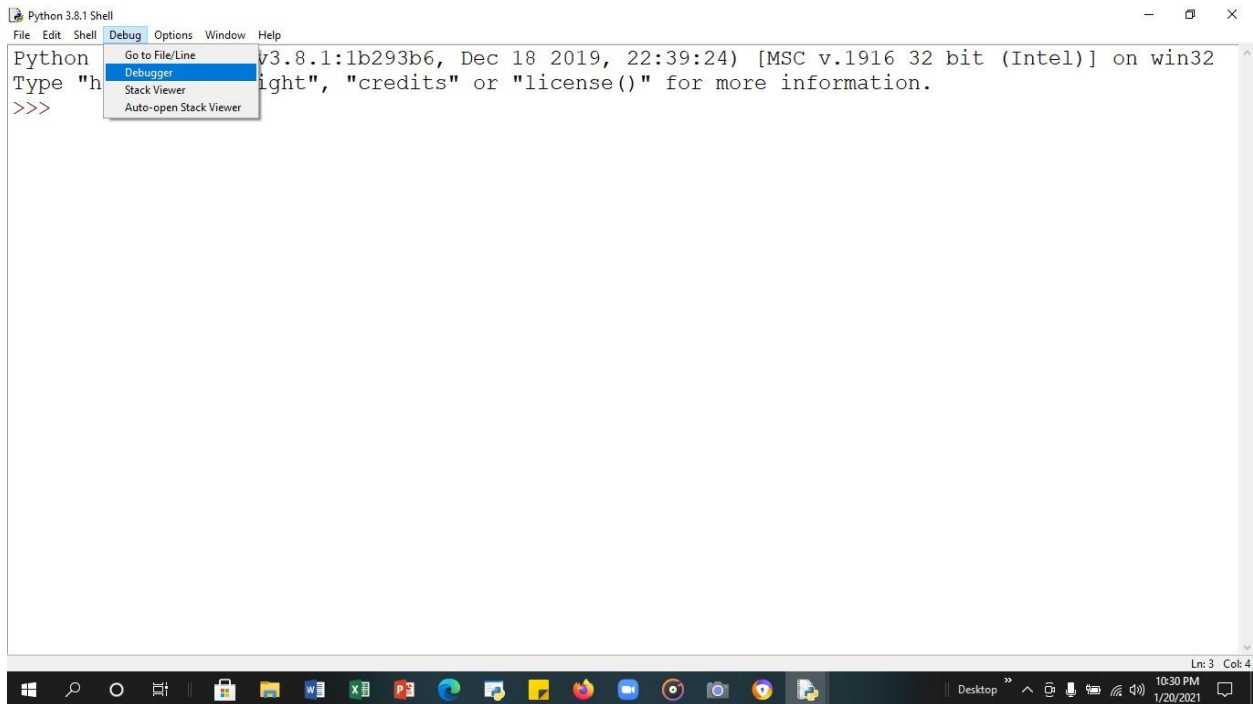
```
print("enter 1st number")
num1=input()
print("enter 2nd number")
num2=input()
print("enter 3rd number")
num3=input()
sum=num1+num2+num3
print("sum is",sum)
```

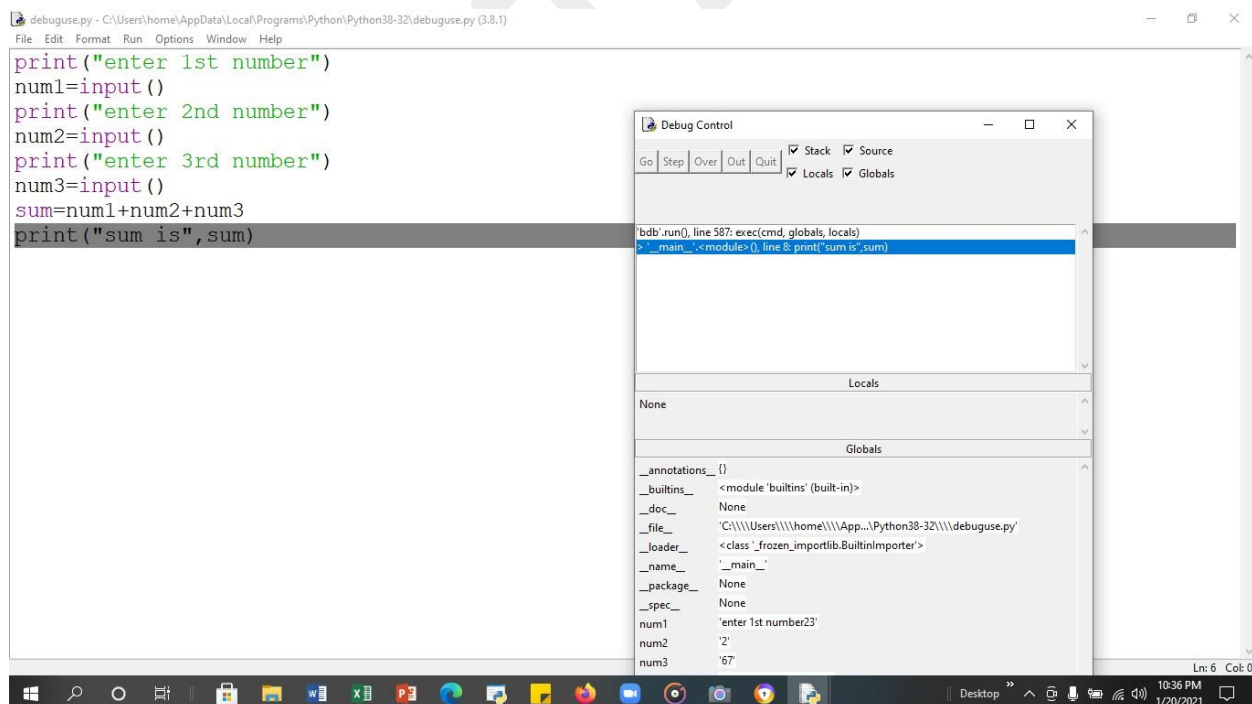
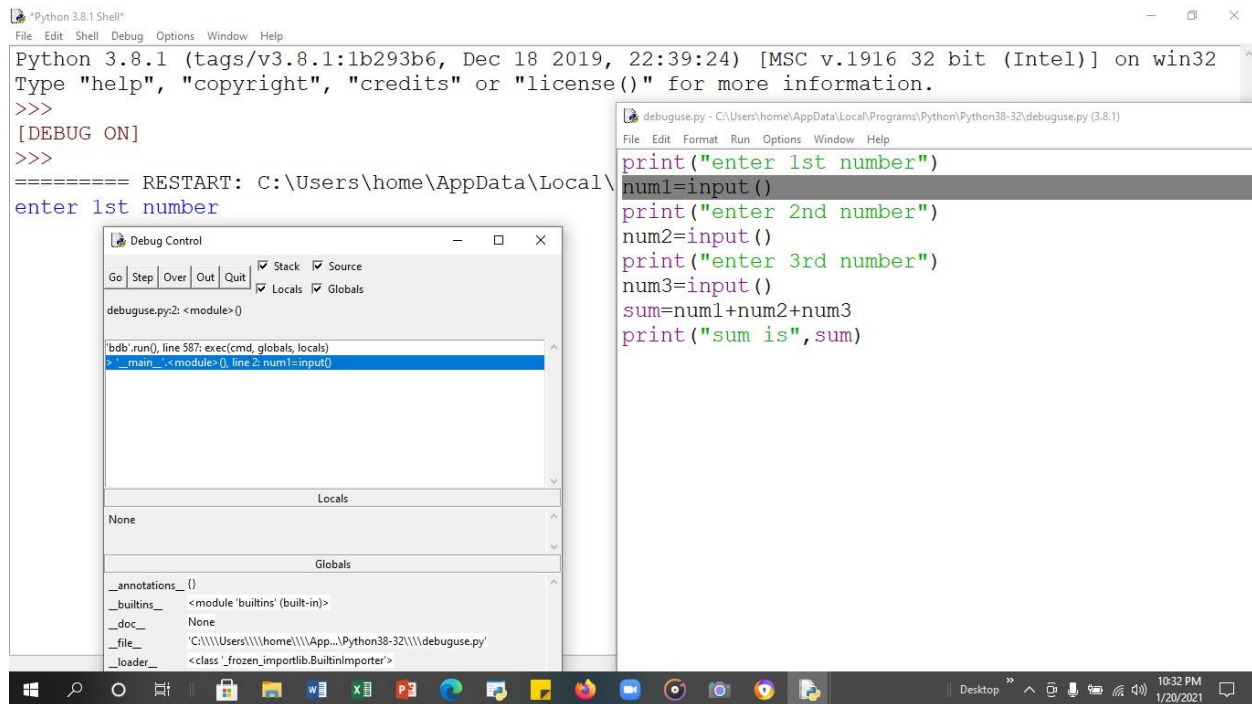
#### without debugger o/p and not able to detect error



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\home\AppData\Local\Programs\Python\Python38-32\debuguse.py =
enter 1st number
12
enter 2nd number
34
enter 3rd number
45
sum is 123445
>>> |
```



**O/p Instead of sum it displayed concatenated values – Resolve using Debugger**



Debugging mode also adds the following buttons to the top of the editor:

Continue, Step Over, Step In, and Step Out. The usual Stop button is also available.



### Continue

Clicking the Continue button will cause the program to execute normally until it terminates or reaches a breakpoint. If you are done debugging and want the program to continue normally, click the Continue button.

### Step In

Clicking the Step In button will cause the debugger to execute the next line of code and then pause again. If the next line of code is a function call, the debugger will “step into” that function and jump to the first line of code of that function.

### Step Over

Clicking the Step Over button will execute the next line of code, similar to the Step In button. However, if the next line of code is a function call, the Step Over button will “step over” the code in the function. The function’s code will be executed at full speed, and the debugger will pause as soon as the function call returns.

### Step Out

Clicking the Step Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step In button and now simply want to keep executing instructions until you get back out, click the Out button to “step out” of the current function call.

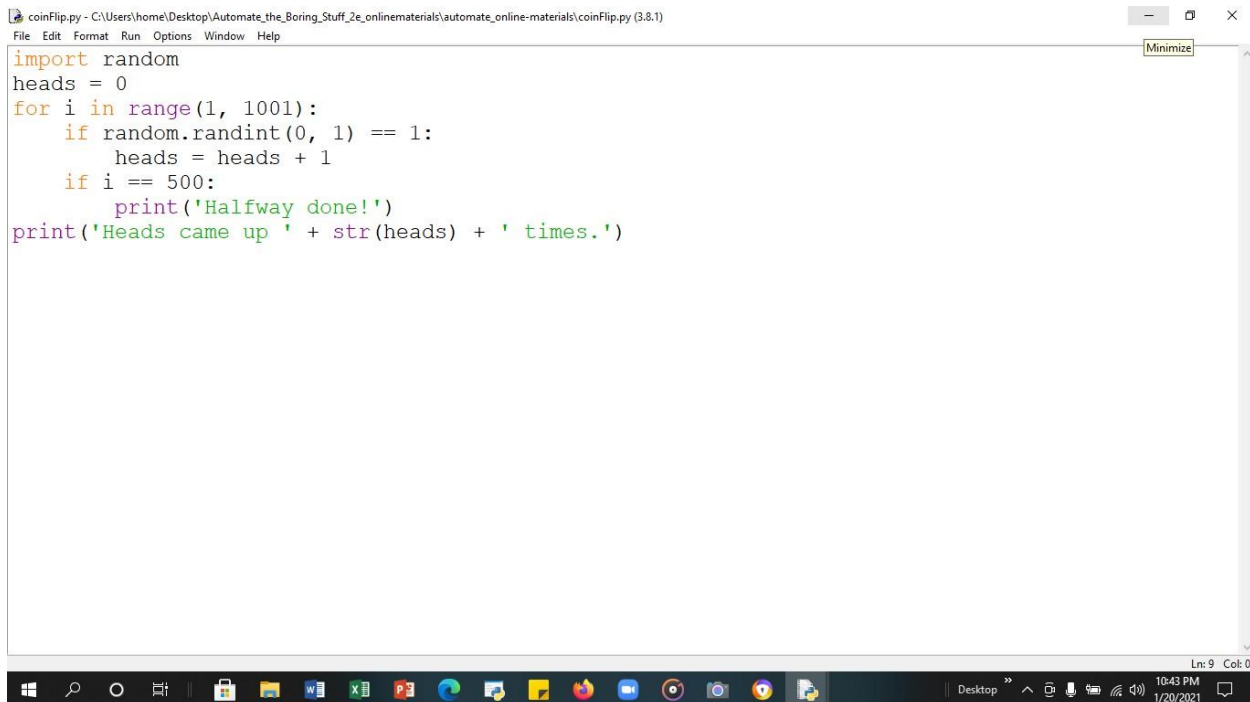
### Stop

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Stop button. The Stop button will immediately terminate the program.

Stepping through the program with the debugger is helpful but can also be slow. Often you’ll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with **breakpoints**.

## Breakpoints

A breakpoint can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor tab and enter the following program, which simulates flipping a coin 1,000 times. Save it as coinflip.py

A screenshot of a Python IDE window titled 'coinFlip.py - C:\Users\home\Desktop\Automate\_the\_Boring\_Stuff\_2e\_onlinematerials\automate\_online-materials\coinFlip.py (3.8.1)'. The window contains the following Python code:

```
import random
heads = 0
for i in range(1, 1001):
    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

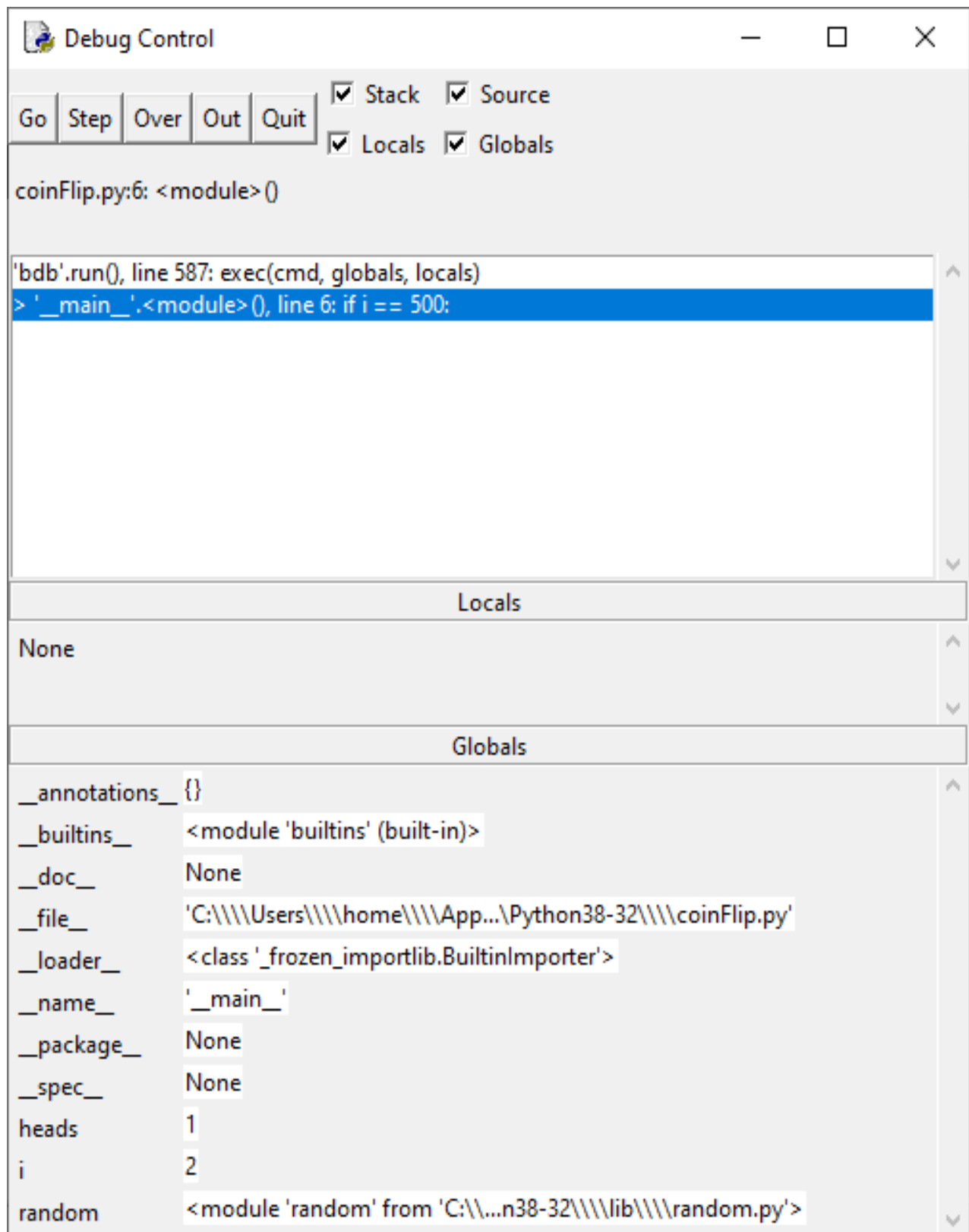
The IDE has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. A 'Minimize' button is visible in the top right corner of the window. The Windows taskbar is visible at the bottom, showing various application icons and the system clock indicating 10:43 PM on 1/20/2021.

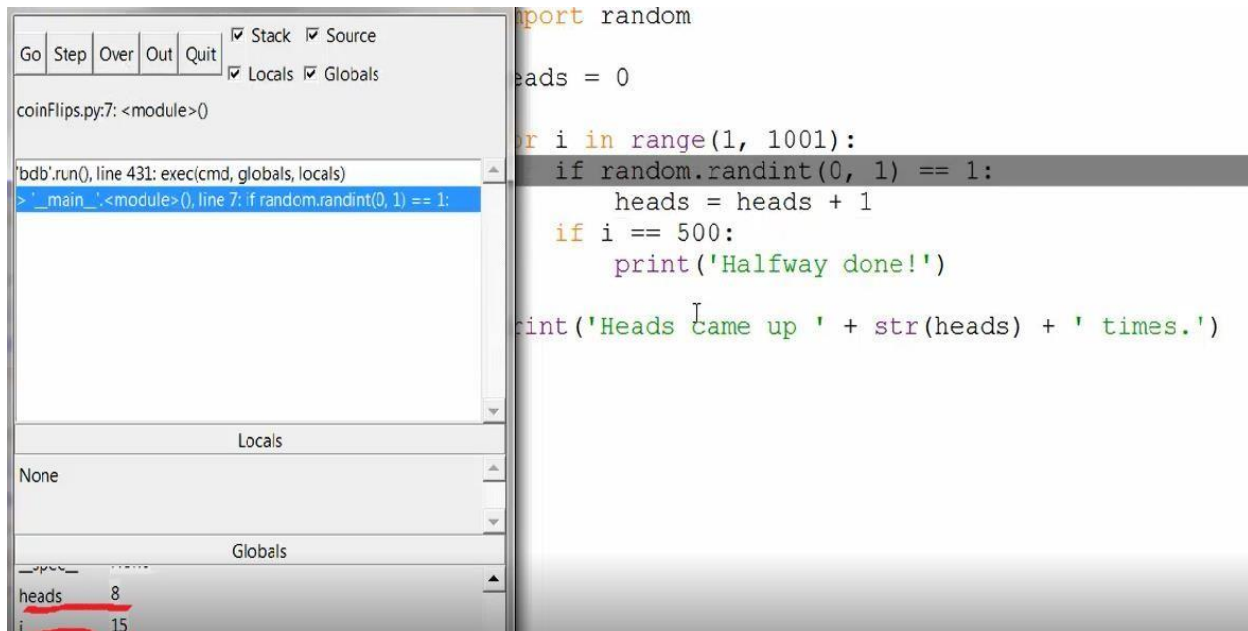
The `random.randint(0, 1)` call ❶ will return 0 half of the time and 1 the other half of the time. This can be used to simulate a 50/50 coin flip where 1 represents heads. When you run this program without the debugger, it quickly outputs something like the following:

**Halfway done!**

**Heads came up 490 times.**

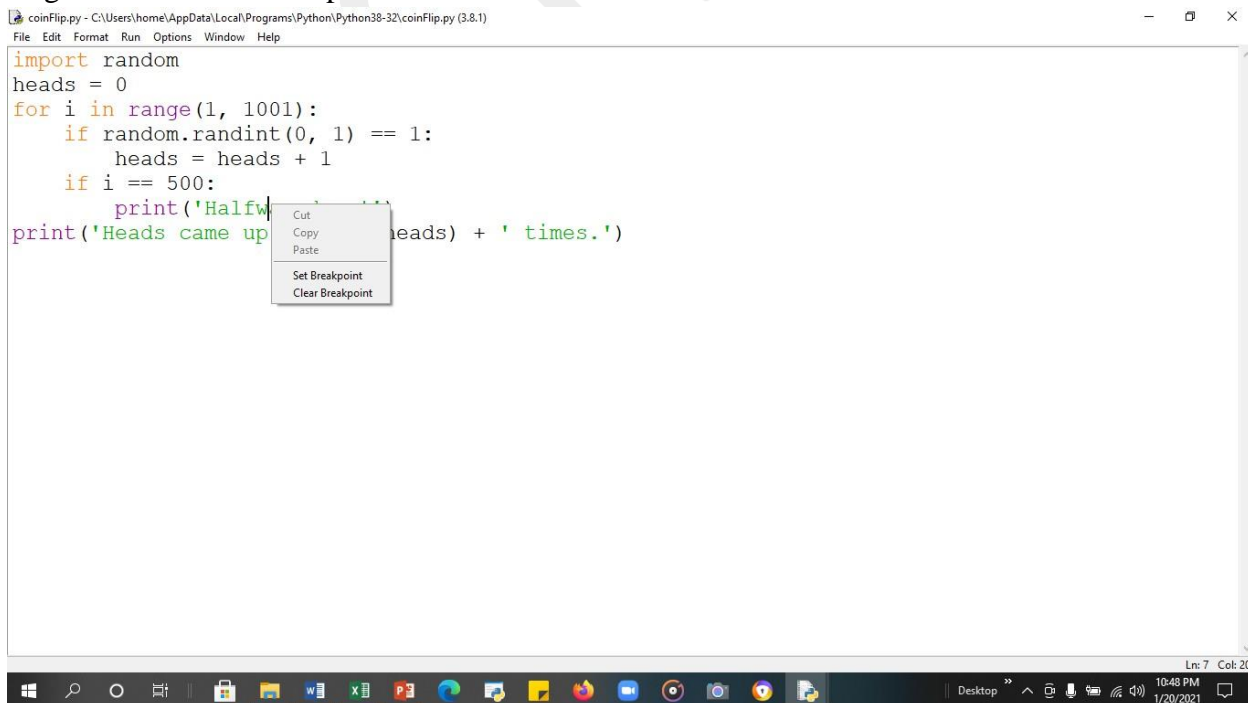
If you ran this program under the debugger, you would have to click the Step Over button thousands of times before the program terminated. If you were interested in the value of heads at the halfway point of the program's execution, when 500 of 1,000 coin flips have been completed,



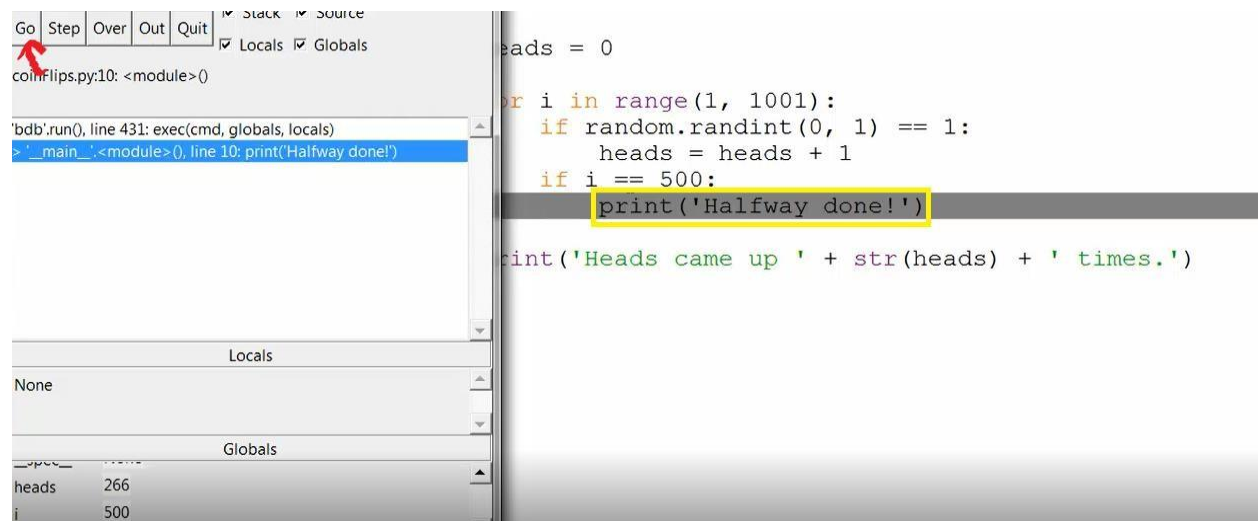


**You could instead just set a breakpoint on the line `print('Halfway done!')`**

②. To set a breakpoint, click the line number in the file editor to cause a red dot to appear, marking the breakpoint. Setting a breakpoint causes line to highlight in yellow color. You don't want to set a breakpoint on the if statement line, since the if statement is executed on every single iteration in the loop.



.The line with the breakpoint highlighted in yellow color . When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Continue, the program will run at full speed until it reaches the line with the breakpoint set on it. You can then click Continue, Step Over, Step In, or Step Out to continue as normal.



When you want to check the line within if block i.e print(“halfway done”) the value of I should be 500 and it results in keep on clicking till i value reached to 500.To overcome it set breakpoint on print statement within if block and click on go button, then i value reaches immediately to 500.Continue execution using over. If you want to remove a breakpoint, click the line number again and the debugger will not break on that line in the future.

## SUMMARY

- ✓ Assertions, exceptions, logging, and the debugger are all valuable tools to find and prevent bugs in your program. Assertions with the Python assert statement are a good way to implement sanity checks that give you an early warning when a necessary condition doesn't hold true. Assertions are only for errors that the program shouldn't try to recover from and should fail fast. Otherwise, you should raise an exception.
- ✓ An exception can be caught and handled by the try and except statements. The logging module is a good way to look into your code while it's running and is much more convenient to use than the print() function because of its different logging levels and ability to log to a text file. The debugger lets you step through your program one line at a time.
- ✓ Alternatively, you can run your program at normal speed and have the debugger pause execution whenever it reaches a line with a breakpoint set. Using the debugger, you can see the state of any variable's value at any point during the program's lifetime. These debugging tools and techniques will help you write programs that work. Accidentally introducing bugs into your code is a fact of life, no matter how many years of coding experience you have.