Lesson 5: String Manipulation

A string is a Sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
fruit = 'banana'
letter = fruit[1]
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an ***index***. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

Code	Result
`fruit = 'banana'` **`letter = fruit[1]`**	a

For most people, the first letter of 'banana' is b, not a. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

Code	Result
`fruit = 'banana'` **`letter = fruit[0]`**	b

So b is the 0th letter ("zero-eth") of 'banana', a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter. Here are all of indices of the characters in the word 'banana':

Index	Character	Using the Bracket Operator
0	b	`fruit[0]`
1	a	`fruit[1]`
2	n	`fruit[2]`
3	a	`fruit[3]`
4	n	`fruit[4]`
5	a	`fruit[5]`

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

Code	Result
`fruit = 'banana'` `letter = fruit[1.5]`	`TypeError: string indices must be integers`

Go online to complete the Indexing Strings Challenge

Getting the length of a String

`len` is a built-in function that returns the number of characters in a string:

Code	Result
`fruit = 'banana'` `length = len(fruit)`	`6`

To get the last letter of a string, you might be tempted to try something like this:

Code	Result

`fruit = 'banana'`
`length = len(fruit)`
`last = fruit[length]`
	`IndexError: string index out of range`

The reason for the `IndexError` is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

Code	Result

`fruit = 'banana'`
`length = len(fruit)`
`last = fruit[length-1]`
	`a`

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

Code	Result
`fruit = 'banana'` `last = fruit[-1]`	`a`

Go online to complete the
Negative Indexes in Strings Challenge

Traversal through a string with a Loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal.

One way to write a traversal is with a while loop:

Code	Output

`fruit = 'banana'`
 	
`index = 0`
 	
`while index < len(fruit):`
`    letter = fruit[index]`
`    print(letter)`
`    index = index + 1`
	b
	a
	n
	a
	n
	a

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Go online to complete the
Using the Length of a String in a while Loop Challenge

Another way to write a traversal is with a for loop:

Code	Output

`fruit = 'banana'`
 	
`for char in fruit:`
`    print(char)`
	b
	a
	n
	a
	n
	a

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

Go online to complete the
Iterating Over a String Using a for Loop Challenge

String Slices

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:

Code	Result
`name = 'Monty Python'` `name[0:5]`	`Monty`
`name[6:12]`	`Python`

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last.

Go online to complete the String Slicing, Part 1 Challenge

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

Code	Output
`fruit = 'banana'` `print(fruit[:3])`	`ban`
`print(fruit[3:])`	`ana`

Go online to complete the String Slicing, Part 2 Challenge

Go online to complete the String Slicing, Part 3 Challenge

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

Code	Output
`fruit = 'banana'` `print(fruit[3:3])`	*(empty string)*

```
print(fruit[4:3])                    (empty string)
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Just like with negative indices, you can use negative numbers to slice backwards from the end of the string. A negative number as the first argument

Code	Output	Notes
`sentence = "Tis but a flesh wound!"` `print(sentence[-2:])`	`d!`	We start 2 characters in from the end and display the remainder of the string.
`print(sentence[-6:-1])`	`wound`	We start 6 characters in from the end of the string and end 1 character from the end of the string.
`print(sentence[:-15])`	`Tis but`	We start at the beginning of the string and end 15 characters from the end of the string.

Go online to complete the String Slicing, Part 4 Challenge

You can also combine positive and negative numbers, though it might get confusing:

Code	Output	Notes
`sentence = "Tis but a flesh wound!"` `print(sentence[10:-1])`	`flesh wound`	We start on the 10th index (the 11th character) and end one character from the end.

Finally, recall the `range` and `randrange` functions. Remember how you could set a `step` argument? Slicing allows you to step as well with an optional third argument. Note carefully how the step argument works. You will always get the first character in your set, then skip ahead n-number of characters to get to the next one.

Code	Output	Notes
`sentence = "Tis but a flesh wound!"` `print(sentence[::2])`	`Tsbtafehwud`	We start at the beginning and we go to the end and we want every 2nd character. The first, 3rd, 5th, etc characters are selected.
`print(sentence[1::3])`	`ib fswn`	We start with index 1 (the second character) and go to the end, then select every 3rd character.

`sentence[-10:-1:4]`	ewd	We start at the 10th character from the end and go to 1st character from the last character, then take every 4th character.

Go online to complete the String Slicing, Part 5 Challenge

Strings are Immutable

It is tempting to use the [ ] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

Code	Output

`greeting = 'Hello, world!'`
 	
`greeting[0] = 'J'`
	`TypeError: object does not support item assignment`

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later. An item is one of the values in a sequence.

The reason for the error is that strings are *immutable*, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

Code	Output

`greeting = 'Hello, world!'`
 	
`new_greeting = 'J'+greeting[1:]`
 	
`print(new_greeting)`
`print(greeting)`
	`Jello, world!`
	`Hello, world!`

This example concatenated a new first letter onto a slice of greeting. It has no effect on the original string.

Looping and Counting

The following program counts the number of times the letter *a* appears in a string:

Code	Output

`word = 'banana'`
 	
`count = 0`
	3

```
for letter in word:
    if letter == 'a':
        count += 1

print(count)
```

This program demonstrates another pattern of computation called a ***counter***. The variable `count` is initialized to 0 and then incremented each time an 'a' is found. When the loop exits, `count` contains the result—the total number of a's.

The in Operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

Code	Result
`'a' in 'banana'`	`True`
`'seed' in 'banana'`	`False`

You can also use the logical operator `not` to check if something is not found in a string:

Code	Result
`'a' not in 'banana'`	`False`
`'seed' not 'banana'`	`True`

Here is how you might use this in code:

Code	Output

`word = 'banana'`
 	
`if 'a' in word:`
`  print('We found an A!')`
`else:`
`  print('There is no A in this word')`
	`We found an A!`

`dessert = 'banana pie'`
 	
`if 'pie' not in dessert:`
`  print('No pie for you!')`
`else:`
`  print('We\'re having pie tonight!')`
	`We're having pie tonight!`

Go online to complete the
Testing Strings, Part 1 Challenge

String Comparison

The comparison operators work on strings.

To see if two strings are equal:

Code

```
if word == 'banana':
    print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

Code

```
if word < 'banana':
    print(word+', comes before banana.')

elif word > 'banana':
    print(word+', comes after banana.')

else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Code

```
Pineapple comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

Concatenating Strings

Concatenation is a fancy word for combining. When we concatenate strings, we are combining them into a single string. We saw way back in Lesson 1 that the + operator can be used to "add" strings together. This is concatenation. You can concatenate strings and variables together:

Code	Output
`fruit = 'banana'` `dessert = 'pie'`	`bananapie`

`print(fruit + dessert)`	

`dessert = 'banana ' + 'pie'`
 	
`print(dessert)`
	`banana pie`

`fruit = 'banana'`
 	
`print(fruit+' pie')`
	`banana pie`

You can also use the shortcut code += to concatenate multiple strings.

Code	Output

`fruit = 'b'`
`fruit += 'a'`
`fruit += 'n'`
`fruit += 'a'`
`fruit += 'n'`
`fruit += 'a'`
 	
`print(fruit)`
	`banana`

`fruit = 'banana'`
 	
`fruit += ' '`
 	
`fruit += 'pie`
 	
`print(fruit)`
	`banana pie`

Go online to complete the Concatentating Strings Challenge

Replicating Strings

We also saw way back in Lesson 1 that the * operator can be used to "multiply" strings.

Code	Output

`greeting = 'Hi!'`
 	
`print(greeting * 5)`
	`Hi!Hi!Hi!Hi!Hi!`

Go online to complete the
The Repetition Operator for Strings Challenge

string Methods

Strings are an example of Python objects. An object contains both data (the actual string itself) and methods, which are effectively functions that are built into the object and are available to any instance of the object.

Python has a function called `dir` which lists the methods available for an object. The type function shows the type of an object and the `dir` function shows the available methods.

Code	Result
`stuff = 'Hello world'` **`type(stuff)`**	`<type 'str'>`

`dir(stuff)`	`['capitalize', 'center',`
	`'count', 'decode',`
	`'encode', 'endswith',`
	`'expandtabs', 'find',`
	`'format', 'index',`
	`'isalnum', 'isalpha',`
	`'isdigit', 'islower',`
	`'isspace', 'istitle',`
	`'isupper', 'join',`
	`'ljust', 'lower',`
	`'lstrip', 'partition',`
	`'replace', 'rfind',`
	`'rindex', 'rjust',`
	`'rpartition', 'rsplit',`
	`'rstrip', 'split',`
	`'splitlines',`
	`'startswith', 'strip',`
	`'swapcase', 'title',`
	`'translate', 'upper',`
	`'zfill']`

Calling a method is similar to calling a function—it takes arguments and returns a value—but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter. A method call is called an *invocation*; in this case, we would say that we are *invoking* upper on the `word`.

Instead of the function syntax `action(word)`, it uses the method syntax `word.action()`.

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

Testing Methods

The testing methods test a string for a condition. These methods are the ones which begin with the word "is" as though you are asking a question. These methods always return a Boolean value - either `True` or `False`. They are very useful in conditional statements and while loops.

The isalnum Method

The isalnum method returns `True` if all characters in the string are alphanumeric (letter and numbers) and there is at least one character in string. Returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.isalnum()`	`True`	All characters in this string are letters and numbers, therefore `isalnum()` returns `True`.
`word = 'Banana 5!'` `word.isalnum()`	`False`	The exclamation point and space are not alphanumeric characters, therefore `isalnum()` returns `False`.

The isalpha Method

The `isalpha` method returns `True` if all characters in the string are alphabetic and there is at least one character in the string, returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.isalpha()`	`True`	All characters in this string are letters, therefore `isalpha()` returns `True`.
`word = 'Banana 5!'` `word.isalpha()`	`False`	The exclamation point, the space, and the number 5 are all not alphabetic characters. Therefore `isalpha()` returns `False`.

The islower Method

The `islower` method returns `True` if all cased characters in the string are lowercase and there is at least one cased character in the string. Returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.islower()`	`True`	All cased-characters (letters) in this string are in lowercase, therefore `islower()` returns `True`.
`word = 'Banana 5!'` `word.islower()`	`False`	The capital B is not a lower case letter, therefore `islower()` returns `False`.

`word = 'banana 5!'` `word.islower()`	`True`	Even though there is a space, a number, and an exclamation point in the string, `islower()` only looks at characters which *can* be cased: letters. This time all letters are in lowercase. Therefore, `islower()` returns `True`.

The isnumeric Method

The `isnumeric` method returns `True` if there are only numeric characters in the string. Returns `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

It should be noted that this does not check whether a string can be converted into a number using `int()` or `float()`. It only checks for Unicode values within the string. This can be confusing.

Code	Result	Notes
`string = 'banana'` `string.isnumeric()`	`False`	All characters in this string are letters, therefore `isnumeric()` returns `False`.
`answer = '42'` `answer.isnumeric()`	`True`	All characters are numbers, therefore `isnumeric()` returns `True`.
`answer = '42.5'` `answer.isnumeric()`	`False`	While we consider `42.5` to be a number, the period is not considered to be a numeral in Unicode. Therefore `isnumeric()` returns `False`.

The isspace Method

The `isspace` method returns `True` if all characters in the string are whitespace and there is at least one character in the string. Returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.isspace()`	`False`	All characters in this string are letters, therefore `isspace()` returns `False`.
`word = 'Banana 5!'` `word.isspace()`	`False`	While there is a space in this string, there are non-space characters too. Therefore `isspace()` returns `False`.
`space = ' '` `space.isspace()`	`True`	The only character in this string is a space, therefore `isspace()` returns `True`.

The istitle Method

The isstitle method returns `True` if the string is a title-cased string and there is at least one character in the string, i.e. upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones. Method returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.istitle()`	`False`	The one word in this string begins with a lowercase letter, therefore `istitle()` returns `False`.
`word = 'Banana 5!'` `word.istitle()`	`True`	The only cased word in this string is Banana which starts with an upper case. Therefore `istitle()` returns `True`.
`word = 'Banana pie'` `word.istitle()`	`False`	There are two cased words in this string, but only one of them begins with a capital letter. Therefore `istitle()` returns `False`.

The isupper Method

The isupper method returns `True` if all cased characters in the string are uppercase and there is at least one cased character in the string. Returns `False` otherwise.

Code	Result	Notes
`string = 'banana'` `string.isupper()`	`False`	The only cased word in this string is banana which is in all lower case. Therefore `isupper()` returns `False`.
`word = 'BANANA 5!'` `word.isupper()`	`True`	The only cased word in this string is BANANA which is in all upper case. Therefore `isupper()` returns `True`.

Go online to complete the
Testing Strings, Part 2 Challenge

Manipulation Methods

The remaining methods which are available to the `str` object make changes to the string. In other words, they manipulate the string. It should be noted that we are not *permanently* replacing the string. When each method is invoked, we are making a COPY of that string and leaving the original string alone.

The upper Method

The method `upper` takes a string and returns a new string with all uppercase letters:

Code	Result
`word = 'banana'` `word.upper()`	BANANA

In this example, `word.upper()` is a copy of word. If we were to display the value of `word`, we would see that it is still the same as it was before we invoked the `upper` method:

Code	Output

`word = 'banana'`
`word.upper()`
 	
`print(word.upper())`
 	
`print(word)`
	BANANA
	banana

See? The variable `word` is left untouched. This is true for all of these methods.

The lower Method

The `lower` method will take a string and return a new string with all lowercase letters:

Code	Output

`word = 'bAnaNa'`
`new_word = word.lower()`
`print(new_word)`
	banana

The capitalize Method

The `captialize` method will take a string and return a new string with the first character capitalized and the remaining letters in lowercase. If the first character is not a letter, then the string is returned in lowercase:

Code	Output

`word = 'bAnaNa'`
`new_word = word.capitalize()`
`print(new_word)`
	Banana

`word = '#1 bAnaNa'`
`new_word = word.capitalize()`
`print(new_word)`
	#1 banana

The strip Method

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the strip method.

Note: *I have included quotes around the string to show the white space in the output.*

Code	Output

`line = ' Here we go '`
`new_line = line.strip()`
`print(new_line)`
	`'Here we go'`

The lstrip Method

Only want to remove white space from the left side of the string? Use the `lstrip` method.

Note: *I have included quotes around the string to show the white space in the output.*

Code	Output

`line = ' Here we go '`
`new_line = line.lstrip()`
`print(new_line)`
	`'Here we go '`

The rstrip Method

Only want to remove white space from the right side of the string? Use the `rstrip` method.

Note: *I have included quotes around the string to show the white space in the output.*

Code	Output

`line = ' Here we go '`
`new_line = line.rstrip()`
`print(new_line)`
	`' Here we go'`

Go online to complete the Modifying Strings Challenge

Searching and Replacing

Now we have some methods which can help us to locate (search) for specific characters or strings, and even replace them with something different.

The find Method

The `find` method searches for the position of one string within another. It returns the index value which represents the beginning of the found string.

Code	Output

`fruit = 'banana'`
`index = fruit.find('a')`
`print(index)`
	`1`

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

The `find` method can find substrings as well as characters:

Code	Output

`fruit = 'banana'`
`index = fruit.find('na')`
`print(index)`
	`2`

It can take as a second argument the index where it should `start`. In this example, we are starting with the 3rd index (or 4th character).

Code	Output

`fruit = 'banana'`
`index = fruit.find('na', 3)`
`print(index)`
	`4`

If the substring is not found, then it will return a value of `-1`.

Code	Output

`fruit = 'banana'`
`index = fruit.find('apple')`
`print(index)`
	`-1`

The find method can also has an optional stop parameter. With `stop`, you can indicate where you wish to stop comparing the string. The numbers used for `start` and `stop` and the index values in the string. You cannot use `stop` without `start`, but you can use `start` by itself. When you use `start` and `stop`, it means "from and **including** the `start` index to, and **excluding**, the `stop` index".

Code	Output	Notes

`word = 'banana'`
`index = word.find('na', 2, 5)`
`print(index)`
	`2`	We are once again starting at the 2nd index, but we're stopping at the 5th index. This means we are only searching the substring `'nan'` (since we are excluding the 6th index). `'nan'` does contain `'na'` therefore `find` returns as the index

		position of the phrase found in the substring as it relates to the original string.
word = 'banana' **index = word.find('na', 4, 5)** **print(index)**	-1	Now we are only looking at the 4th index which is 'n'. Obviously, 'na' is not found in 'n' so a -1 is returned.

The replace Method

The `replace` method will return a copy of the string with all occurrences of old substring replaced by new. By default it will replace all occurrences.

Code	Result	Notes
fruit = 'banana' **print(fruit.replace('a', 'i'))** **print(fruit)**	binini banana	We replaced all instances of the letter 'a' in 'banana' with the letter 'i'. Notice that we did not overwrite the original `fruit` variable.

If the optional argument `count` is given, only the first `count` occurrences are replaced.

Code	Result	Notes
fruit = 'banana' **print(fruit.replace('a', 'i', 2))**	binina	We set `count=2`, so only the first two instances of the letter 'a' were replaced with the letter 'i'

As with the `find` method, the `replace` method can also find and replace substrings as well as characters.

Code	Output
fruit = 'banana' **print(fruit.replace('a', '123', 2))**	b123n123na
print(fruit.replace('a', '\nhi\n', 2))	b hi n hi na

The startswith Method

While this one does not begin with the word "is" it still returns a Boolean value.

The `startswith` method returns `True` if the string starts with the specified prefix, `False` otherwise.

Code	Result	Notes

`line = 'Please have a nice day'` `line.startswith('Please')`	`True`	The string begins with 'Please', therefore `startswith` returns as `True`.
`line = 'Please have a nice day'` `line.startswith('p')`	`False`	The string begins with a capital P, not a lowercase 'p' – therefore `startswith` returns as `False`.

You will note that `startswith` requires the case to match!

The `startswith` method has two optional arguments: `start` and `stop`. With `start`, you can indicate where in the string you wish to start searching. With `stop`, you can indicate where you wish to stop comparing the string. The numbers used for `start` and `stop` and the index values in the string. You cannot use `stop` without `start`, but you can use `start` by itself.

When you use `start` and `stop`, it means "from and **including** the `start` index to, and **excluding**, the `stop` index".

Code	Result	Notes
`line = 'Please have a nice day'` `line.startswith('have', 7)`	`True`	The `start` parameter is set at 7, so we start looking at the 7 index of the string. The `stop` parameter is not set, so the default is to look until the end of the string. This means we are looking for the substring `'have a nice day'` to start with the string `'have'`. It does, therefore `startswith` returns as `True`.
`line = 'Please have a nice day'` `line.startswith('have', 7, 9)`	`False`	We are once again starting at the 7th index, but we're stopping at the 9th index. This means we are only searching the substring `'ha'` (since we are excluding the 9th index). `'ha'` does not begin with `'have'` therefore `startswith` returns as `False`.

The endswith Method

The `endswith` method returns `True` if the string ends with the specified prefix, `False` otherwise.

Code	Result	Notes
`file = 'Lesson 4 Lab.py'` `file.endswith('.py')`	`True`	The string ends with `'.pyc'`, therefore `endswith` returns as `True`.
`file = 'Lesson 4 Lab.py'` `file.endswith('.PY')`	`False`	The string ends with a lowercase `'.py'`, not an upper-case `'.PY'` – therefore `endswith` returns as `False`.

You can also use an optional `start` and `stop` parameters just like with `startswith` to narrow your full string down to a substring:

Code	Result	Notes
`file = 'Lesson 4 Lab.py'` `file.endswith('.py', 9)`	`True`	The `start` parameter is set at 9, so we start looking at the 9 index of the string. The `stop` parameter is not set, so the default is to look until the end of the string. This means we are looking for the substring `'Lab.py'` to end with the string `'.py'`. It does, therefore `endswith` returns as `True`.
`file = 'Lesson 4 Lab.py'` `file.endswith('.py', 9, 12)`	`False`	We are once again starting at the 9th index, but we're stopping at the 12th index. This means we are only searching the substring `'Lab'` (since we are excluding the 12th index). `'Lab'` does not end with `'.py'` therefore `endswith` returns as `False`.

Go online to complete the
Searching and Replacing Strings Challenge

Chaining Multiple Methods

There are times when you need to call multiple methods on a single string. You can do that line by line:

Line	Code	Result
1	`line = 'Please have a nice day'`	`Please have a nice day`
2	`line.lower()`	`please have a nice day`
3	`line.startswith('p')`	`True`

In the last example, the method `lower` is called and then we use `startswith` to see if the resulting lowercase string starts with the letter "p".

As long as we are careful with the order, we can make multiple method calls in a single expression (look closely at line 2 below).

Line	Code	Result
1	`line = 'Please have a nice day'`	`Please have a nice day`
2	`line.lower().startswith('p')`	`True`

Line 2 in the above example combined Lines 2 and 3 in the previous example into a single statement. This is called ***chaining***.

Example: Parsing Strings using find

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

```
From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008
```

and we wanted to pull out only the second half of the address (i.e., `uct.ac.za`) from each line, we can do this by using the find method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space after the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for. Start with this line of code:

```
data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
```

Now, let's try to only get the domain of the email address:

Code	Output	Notes
`start_pos = data.find('@')`		First we use the `find` method to get the index value of the @ sign. We assign this index value to a variable called `start_pos` (for start position).
`print(start_pos)`	`21`	When we display the value of `start_pos`, we get 21. Since indices begin with the number 0, this means that the @ sign is the 21st index or 22nd character of the string.
`end_post = data.find(' ', start_pos)`		Now we want to find where the domain ends. The first character after the domain is a space, so we can search for that. But there is a space before the email address. We use the second, optional argument of start, to pass a starting position. In this case, we want to start where the "@" was found.
`print(end_post)`	`31`	When we display the value of `end_pos`, we get 31. Since indices begin with the number 0, this means that the first space after the @ sign is the 31st index or 32nd character of the string.

domain = data[start_pos+1:end_pos]		Now we can slice our string from the starting index, `start_pos`, + 1 (since we don't actually need the @ sign) to and excluding the ending index of `end_pos`.
print(domain)	`uct.ac.za`	When we display the value of `domain`, we get `uct.ac.za` as expected.

Splitting Strings

Finally, we can separate a single string into multiple strings using the `split` method. The split method returns a list of the words or substrings in the string, using `sep` as the delimiter string. If `sep` is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

Code	Result	Notes

file = 'Lesson 4 Lab.py'
words = file.split()
print(words)
	`['Lesson', '4', 'Lab.py']`	We did not specify `sep`, so it defaulted to splitting the string using spaces. We now have a list with these items:
		
		`Lesson`
		`4`
		`Lab.py`

file = 'Lesson 4 Lab.py'
words = file.split('.')
print(words)
	`['Lesson 4 Lab', 'py']`	This time we specified `sep`, We chose to split this string using the period as the delimiter. We now have a list with these items:
		
		`Lesson 4 Lab`
		`py`

A second argument, `maxsplit`, allows us to only split the string a certain number of times. Let's say we only wanted to pull the first word out. We would set `maxsplit` equal to 1 so only one split occurs. Note that if you are going to use `maxsplit`, you must also explicitly set `sep`, even if you want to use its default value.

Code	Result	Notes
file = 'Lesson 4 Lab.py' **words = file.split(' ', 1)** **print(words)**	`['Lesson', '4 Lab.py']`	We only want to split this string once, so we specify `maxsplit` to equal 1. Because we're using `maxsplit`, we have to set the

first argument sep. We now have a list with these items:

```
Lesson
4 Lab.py
```

Go online to complete the Splitting Strings Challenge

Exception Handling

We saw way back in Lesson 1 that a user can cause a bit of chaos with your program. If your code prompts them for a number and they enter a letter, then you try to convert that letter into a number, an error will be thrown:

```
>>> age = input('How old are you'?)
How old are you? Too old!

>>> int(age)
ValueError: invalid literal for int() with base 10
```

Your script immediately stops in its tracks with a *traceback*. It does not continue and your user is left wondering what happened.

In your practice coding over the last few units, you have probably thrown several unintentional errors yourself. Here is a list of common ones in case you missed them:

Exception	Meaning
`IndexError`	Raised when a sequence subscript is out of range.
`NameError`	Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.
`OSError`	This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).
`RuntimeError`	Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
`ValueError`	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

Here is a sample program which asks a user to enter how many pieces of pie they want, then displays what percentage of the whole pie they will get:

```
pieces = input("How many pieces of pie do you want? ")
percentage = 1/int(pieces)
print("You get", format(percentage, ".2%"), "of the pie!")
```

If we execute this code and give it good code, it the output looks something like this:

```
How many pieces of pie do you want? 2
That's 50.00% of the pie!
```

If we execute the code and give it invalid input, it simply fails with an unfriendly error message:

```
How many pieces of pie do you want? one

Traceback (most recent call last):
  File " example.py", line 1, in <module>
    pieces = int(input("How many pieces of pie do you want? "))
ValueError: invalid literal for int() with base 10: 'one'
```

And the program just stops.

We need a better way to handle these exceptions and allow the program to correct the issue or skip the code which is affected.

The try / except Statement

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "`try / except`".

The idea of try and except is that you know that there is the potential for some sequence of instruction(s) to have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the `except` block) are ignored if there is no error.

You can think of the try and except feature in Python as an "insurance policy" on a sequence of statements.

We can rewrite our temperature converter as follows:

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except:
    print("You need to enter a number!")
```

How does this work? Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block. The `except` block is known as the ***handler*** because it handles the exception.

```
How many pieces of pie do you want? One
You need to enter a number!
```

Handling an exception with a `try` statement is called ***catching*** an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Go online to complete the
Exception Handling, Part 1 Challenge

Displaying Exceptions

Sometimes we want to see the exception being thrown. Going back to our example with pie: what if we enter a zero for our input?

```
How many pieces of pie do you want? 0
You need to enter a number!
```

What happened here? A zero IS a valid number. Why is an error being thrown? We can do a couple of things to figure out which error was caught by the `except` block.

First, you assign the exception to a variable like this:

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except Exception as my_exception:
    print(my_exception)
    print("You need to enter a number!")
```

Here we have explicitly called on the `Exception` object and assigned it to the variable, `my_exception`. This results in this output:

```
How many pieces of pie do you want? 0
division by zero
You need to enter a number!
```

Ah ha! Our problem is related to division by 0.

Go online to complete the
Exception Handling, Part 2 Challenge

The second way you can figure out which error was thrown is to the `raise` statement. The `raise` statement allows you to intentionally "raise" an exception. It can take a specific exception type as an argument, but when it is just called by itself, it will display the most recently thrown error. *Note: the program will stop once the raise statement is called. This is best used only when troubleshooting.*

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except:
    raise
    print("You need to enter a number!")
```

Now when we enter a zero, we get this:

```
How many pieces of pie do you want? 0

Traceback (most recent call last):
  File "example.py", line 3, in <module>
    percentage = 1/int(pieces)
ZeroDivisionError: division by zero
```

Our second way also narrowed it down to a division by zero error - or `ZeroDivisionError`.

Specifying the Exception

Obviously, telling the end user that they need to enter a number when zero IS a valid number is not a good solution. We need special messages depending on the situation. Fortunately, Python has you covered.

A `try` statement may have more than one `except` clause, to specify handlers for different exceptions. This is useful for when you want to handle one type of error one way and a different type of error a different way.

Here's how we use it in the pie example:

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except ValueError:
    print("You need to enter a number!")

except ZeroDivisionError:
    print("You cannot enter a zero!")
```

Notice how the exception type is specified immediately after the `except` declaration. When we run this code, here are our outputs:

```
How many pieces of pie do you want? 2
You get 50.00% of the pie!

How many pieces of pie do you want? One
You need to enter a number!

How many pieces of pie do you want? 0
You cannot enter a zero!
```

Now both of our known exceptions are covered.

When you have multiple except handlers, at most one handler will be executed.

You can also assign the contents of the error to a variable for use within the `except` block:

```
except ValueError as my_error:

except ZeroDivisionError as division_error:
```

Go online to complete the
Exception Handling, Part 3 Challenge

Specifying Multiple Exceptions at Once:

A single `except` clause may handle multiple types of exceptions:

```
except (NameError, ValueError, TypeError):
```

This is useful (and more efficient) if you want to handle different exception types the same way.

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except (ValueError, TypeError):
    print("You need to enter a number!")

except ZeroDivisionError:
    print("You cannot enter a zero!")
```

In this example, when `ValueError` or `TypeError` are thrown, they will both be caught by the same handler and `You need to enter a number!` will display.

Default Handling

Going back to our pie example, we are now handling very specific error types, but it is good practice to include the default except block in case something unexpected occurs.

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)
    print("You get", format(percentage, ".2%"), "of the pie!")

except (NameError, ValueError, TypeError):
    print("You need to enter a number!")

except ZeroDivisionError:
    print("You cannot enter a zero!")

except:
    print("Something bad happened. No pie for you.")
```

It should be noted that handlers only handle exceptions that occur in the corresponding `try` clause, not in other handlers of the same `try` statement. This means that if you raise a second exception while handling the first exception, your original `try / except` statement will not be able to handle the second exception. You would have to create a new, nested `try / except` statement within your exception handing.

Go online to complete the
Exception Handling, Part 4 Challenge

else Clause

The `try / except` statement has an optional `else` clause. The `else` clause comes after all `except` clauses and will only execute if the `try` clause does not raise an exception.

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try / except` statement.

Let's go back to pie!

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)

except (NameError, ValueError, TypeError):
    print("You need to enter a number!")

except ZeroDivisionError:
    print("You cannot enter a zero!")

except:
    print("Something bad happened. No pie for you.")

else:
    print("You get", format(percentage, ".2%"), "of the pie!")
```

In this case, we anticipate that the line `percentage = 1/int(pieces)` might throw errors if the input is not an integer or if it is equal to 0. This is the only code which we want to catch our specific errors. So, we move the line `print("You get", format(percentage, ".2%"), "of the pie!")` into the else clause. The output looks identical:

```
How many pieces of pie do you want? 2
You get 50.00% of the pie!

How many pieces of pie do you want? One
You need to enter a number!

How many pieces of pie do you want? 0
You cannot enter a zero!
```

However, our program will now run cleaner with the extra code placed in the `else` clause.

Go online to complete the
Exception Handling, Part 5 Challenge

Clean-up Actions (the finally Clause)

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed.

The `finally` clause is also executed "on the way out" when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement.

```
pieces = input("How many pieces of pie do you want? ")

try:
    percentage = 1/int(pieces)

except (NameError, ValueError, TypeError):
    print("You need to enter a number!")

except ZeroDivisionError:
    print("You cannot enter a zero!")

except:
    print("Something bad happened. No pie for you.")

else:
    print("You get", format(percentage, ".2%"), "of the pie!")

finally:
    print("Thank you for using the Pie Program!")
```

This outputs like this:

```
How many pieces of pie do you want? 2
You get 50.00% of the pie!
Thank you for using the Pie Program!

How many pieces of pie do you want? 0
You cannot enter a zero!
Thank you for using the Pie Program!

How many pieces of pie do you want? One
You need to enter a number!
Thank you for using the Pie Program!
```

As you can see, the `finally` clause is executed in any event.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

Go online to complete the
Exception Handling, Part 6 Challenge

So here is our full, annotated try / except

```
# Get input from the user
pieces = input("How many pieces of pie do you want? ")

# Begin the try / except statement
try:
    # This is the line which might throw an exception
    percentage = 1/int(pieces)

# This will catch NameError,
# ValueError, and TypeError exceptions
except (NameError, ValueError, TypeError):
    # This displays if one of these errors are thrown
    print("You need to enter a number!")

# This will catch ZeroDivisionError errors
except ZeroDivisionError:
    # This displays if ZeroDivisionError is thrown
    print("You cannot enter a zero!")

# This catches anything else which might go wrong
except:
    print("Something bad happened. No pie for you.")

# This code runs if no exception is thrown
else:
    print("You get", format(percentage, ".2%"), "of the pie!")

# This code runs regardless of whether an exception is thrown
finally:
    print("Thank you for using the Pie Program!")
```

Debugging

A skill that you should cultivate as you program is always asking yourself, "What could go wrong here?" or alternatively, "What crazy thing might our user do to crash our (seemingly) perfect program?"

For example, look at the program which we used to demonstrate the while loop in the chapter on iteration:

```
while True:
    line = input('Enter something: ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

Look what happens when the user enters an empty line of input:

```
Enter something: Hello!
Hello!

Enter something: # Don't print this

Enter something:

Traceback (most recent call last):

  File "C:\Users\hcrites\AppData\Local\Programs\Python\Python35-
32\examples.py", line 3, in <module>

    if line[0] == '#' :

IndexError: string index out of range
```

The code works fine until it is presented an empty line. Then there is no zero-th character, so we get a `traceback`. There are two solutions to this to make line three "safe" even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

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

Another way is to safely write the `if` statement using the guardian pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
if len(line) > 0 and line[0] == '#' :
```

Finally, if we are not sure what all could go wrong, the program could be placed within a try / except statement:

```
while True:
    line = input('Enter something: ')
    try:
        if line[0] == '#' :
            continue
    except:
```

```
        line = input('Enter something: ')

    else:
        if line == 'done':
            break
        print(line)
print('Done!')
```