Week4

Video 1:More str Operators

String Comparisons

The equality and inequlity operators can be applied to strings:

```
>>> 'a' == 'a'
True
>>> 'ant' == 'ace'
False
>>> 'a' == 'b'
False
>>> 'a' != 'b'
True
```

We can compare two strings for their dictionary order, comparing them letter by letter:

```
>>> 'abracadabra' < 'ace'
True
>>> 'abracadabra' > 'ace'
False
>>> 'a' <= 'a'
True
>>> 'A' < 'B'
True
```

Capitalization matters, and capital letters are less than lowercase letters:

```
>>> 'a' != 'A'
True
>>> 'a' < 'A'
False
Every letter can be compared:
>>> ',' < '3'
True
```

We can compare a string and an integer for equality:

```
>>> 's' == 3
False
```

We can't compare values of two different types for ordering:

```
>>> 's' <= 3
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>>
TypeError: unorderable types: str() <= int()
```

Testing For Substrings

The operator `in` checks whether a string appears anywhere inside another one (that is, whether a string is a substring of another).

```
>>> 'c' in 'aeiou'
False
>>> 'cad' in 'abracadabra'
True
>>> 'zoo' in 'ooze'
False
```

String length: function `len`

The builtin function len returns the number of characters in a string:

```
>>> len('')
0
>>> len('abracadabra')
11
>>> len('Bwa' + 'ha' * 10)
23
```

Summary

Description	Operator	Example	Result of example
equality	==	'cat' == 'cat'	True
inequality	!=	'cat' != 'Cat'	True
less than	<	'A' < 'a'	True
greater than	>	'a' > 'A'	True
less than or equal	<=	'a' <= 'a'	True
greater than or equal	>=	'a' >= 'A'	True
contains	in	'cad' in 'abracadabra'	True
length of str s	len(s)	len("abc")	3

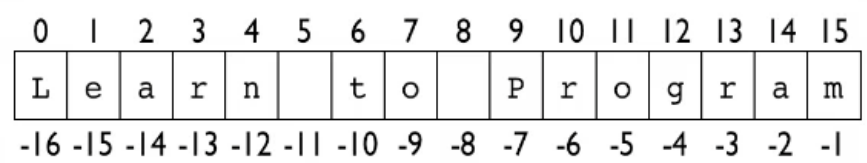
Optional Book Reading

Chapter 5.1 A Boolean Type

Vieo 2: `str`: indexing and slicing

Indexing

An index is a position within the string. Positive indices count from the left-hand side with the first character at index 0, the second at index 1, and so on. Negative indices count from the right-hand side with the last character at index -1, the second last at index -2, and so on. For the string `"Learn to Program"`, the indices are:



The first character of the string is at index 0 and can be accessed using this bracket notation:

```
>>> s[0]
'L'
>>> s[1]
'e'
```

Negative indices are used to count from the end (from the right-hand side):

```
>>> s[-1]
'm'
>>> s[-2]
```

```
'a'
```

Slicing

We can extract more than one character using slicing. A slice is a substring from the start index up to but not including the end index. For example:

```
>>> s[0:5]
'Learn'
>>> s[6:8]
'to'
>>> s[9:16]
'Program'
```

More generally, the end of the string can be represented using its length:

```
>>> s[9:len(s)]
'Program'
```

The end index may be omitted entirely and the default is len(s):

```
>>> s[9:]
'Program'
```

Similarly, if the start index is omitted, the slice starts from index 0:

```
>>> s[:]
'Learn to Program'
>>> s[:8]
'Learn to'
```

Negative indices can be used for slicing too. The following three expressions are equivalent:

```
>>> s[1:8]
'earn to'
>>> s[1:-8]
'earn to'
>>> s[-15:-8]
'earn to'
```

Modifying Strings

The slicing and indexing operations do not modify the string that they act on, so the string that `s` refers to is unchanged by the operations above. In fact, we cannot change a string. Operations like the following result in errors:

```
>>> s[6] = 'd'
Traceback (most recent call last):
    File <"pyshell#19", line 1, in <module>
            s[6] = 'd'
TypeError: 'str' object does not support item assignment
```

Imagine that we want to change string s to refer to `'Learned to Program'`. The following expression evaluates to that `'Learned to Program': s[:5] + 'ed' + s[5:]`

Variable `s` gets the new string: `s = s[:5] + 'ed' + s[5:]`

Notice that the string that `s` originally referred to was not modified: strings cannot be modified. Instead a new string was created and `s` was changed to point to that string.

Video 3: `str` Methods: Functions Inside Objects

Methods

A *method* is a function inside of an object.

The general form of a method call is:

`object.method(arguments)`

String Methods

Consider the code:

```
>>> white_rabbit = "I'm late! I'm late! For a very important date!"
```

To find out which methods are inside strings, use the function dir:

```
>>> dir(white_rabbit)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__','__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Passing `str` as an argument to `dir` gets the same result:

```
>>> dir(str)
```

For many of the string methods, a new string is returned. Since strings are immutable, the original string is unchanged. For example, a lowercase version of the str that white_rabbit refers to is returned when the method lower is called:

```
>>> white_rabbit.lower()
>>> "i'm late! i'm late! for a very important date!"
>>> white_rabbit
>>> "I'm late! I'm late! For a very important date!"
```

To get information about a method, such as the `lower` method, do the following:

```
>>> help(str.lower)
```

Optional Book Reading

Chapter 7.1 Modules, Classes, and Methods
Chapter 7.2 Calling Methods the Object-Oriented Way
Chapter 7.3 Exploring String Methods

Video 4: `for` loop over `str`

For Loops

The general form of a for loop over a string is:

```
for variable in str:
    body
```

The variable refers to each character of the string in turn and executes the body of the loop for each character. For example:

```
>>> s = 'yesterday'
>>> for char in s:
...     print(char)
```

```
...
y
e
s
t
e
r
d
a
y
```

Accumulator pattern: numeric accumulator

Consider the code below, in which the variable `num_vowels` is an accumulator:

```
def count_vowels(s):
    """ (str) -> int

    Return the number of vowels in s. Do not treat letter y as a vowel

    >>> count_vowels('Happy Anniversary!')
    5
    >>> count_vowels('xyz')
    0
    """

    num_vowels = 0

    for char in s:
        if char in 'aeiouAEIOU':
            num_vowels = num_vowels + 1

    return num_vowels
```

The `loop` in the function above will loop over each character that `s` refers to, in turn. The body of the loop is executed for each character, and when a character is a vowel, the `if` condition is `True` and the value that `num_vowel`s refers to is increased by one.

The variable `num_vowels` is an accumulator, because it accumulates information. It starts out referring to the value `0` and by the end of the function it refers to the number of vowels in `s`.

Accumulator pattern: string accumulator

In the following function, the variable `vowels` is also an accumulator:

```
def collect_vowels(s):
    """ (str) -> str

    Return the vowels from s.  Do not treat the letter
    y as a vowel.

    >>> collect_vowels('Happy Anniversary!')
    'aAiea'
    >>> collect_vowels('xyz')
    ''
    """

    vowels = ''

    for char in s:
        if char in 'aeiouAEIOU':
            vowels = vowels + char

    return vowels
```

Variable `vowels` initially refers to the empty string, but over the course of the function it accumulates the vowels from `s`.

Optional Book Reading

Chapter 9.2 Processing Characters in Strings

Video 5: IDLE's Debugger

Debug Control

The Python Visualizer has limitations: it does not allow import statements, and it stops tracing after 300 steps. IDLE comes with a debugger, which is a tool that works a lot like the visualizer but without the pretty pictures. To run a program in IDLE's debugger, the steps are:

1. Make sure the Python Shell window is on top and select Debug->Debugger. This opens a window called "Debug Control".
2. Check the checkbox for Source.
3. Open the Python file where you have saved your program.
4. Select Run->Run Module. This will change the contents of Debug Control.

Understanding The Debug Window

`Step` is like `Forward` in the visualizer: it executes the current instruction. We click `Step` to "step" through the program. The Debug Control window will highlight the current line being executed. It will also show the current variables in the "Locals" pane. The middle pane shows the current stack frames and the current lines of code for each. We can switch back and forth between them to see the variables.