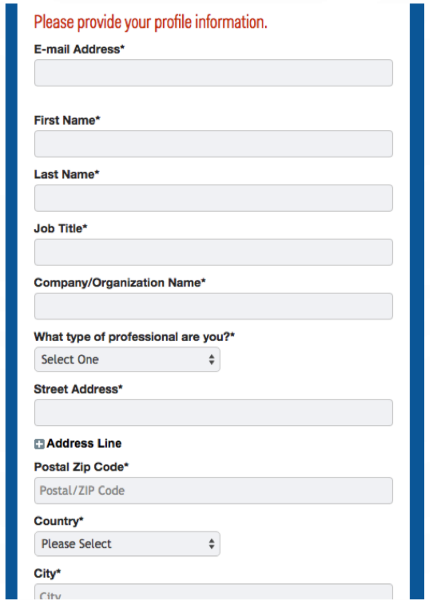
CHAPTER 8
MORE ON STRINGS AND SPECIAL METHODS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING USING PYTHON, LIANG (PEARSON 2013)

MOTIVATIONS

- Text is everywhere! Computers have to interface with humans.
- How could we determine of a tweet has sensitive or false information?
- When filling out a form how do we know a date is a valid format?
  - DD/MM/YYYY
  - MM/DD/YY
  - etc


Please provide your profile information.

E-mail Address*

First Name*

Last Name*

Job Title*

Company/Organization Name*

What type of professional are you?*
Select One

Street Address*

Address Line

Postal Zip Code*
Postal/ZIP Code

Country*
Please Select

City*
City

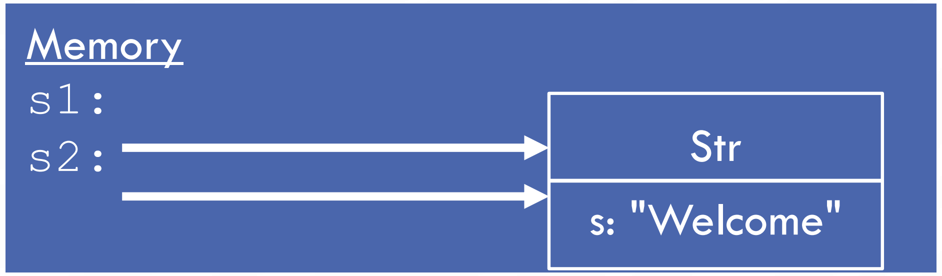
THE STRING CLASS - STR

- Strings are objects that can be constructed and used (invoking methods)
- Creating Strings
  - `s1 = str()           # Create an empty string ("")`
  - `s2 = str("Welcome") # Create a string "Welcome"`
- Python provides a simple syntax for creating string using a string literal. For example:
  - `s1 = ""          # Same as s1 = str()`
  - `s2 = "Welcome" # Same as s2 = str("Welcome")`

STRINGS ARE IMMUTABLE

- A string object is immutable. Once it is created, its contents cannot be changed. To optimize performance, Python uses one object for strings with the same contents. Below, both s1 and s2 refer to the same string object.

```python
s1 = "Welcome"
s2 = "Welcome"
print(id(s1))
print(id(s2))
```



Output

```
505408902
505408902
```

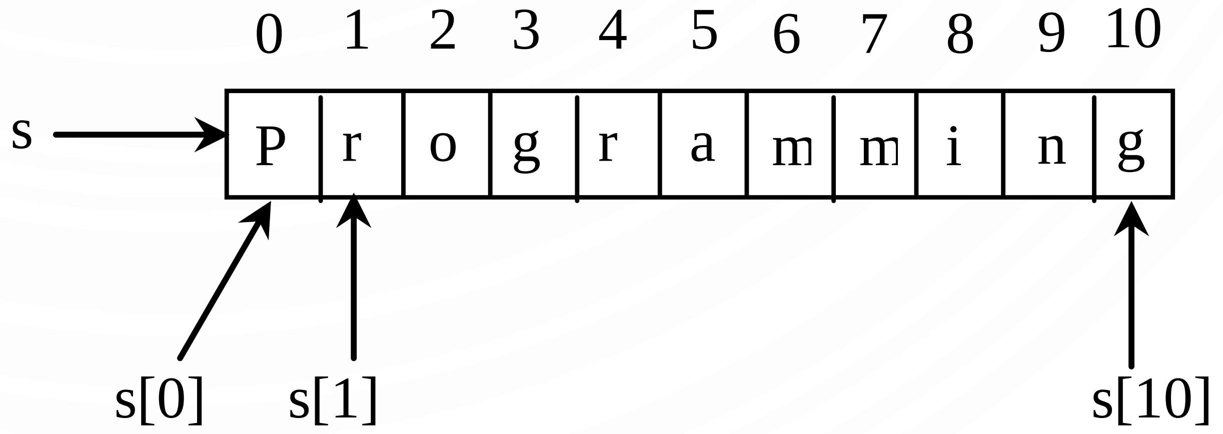
SOME FUNCTIONS WITH STR

- Let a variable `s = "Welcome"`
- `len(s)` – returns the length of the string, in this example 7.
- `max(s)` – returns the character with the largest Ascii value, 'o' in this case
- `min(s)` – returns the character with the smallest Ascii value, 'W' in this case

INDEX OPERATOR []

- Strings are laid out as a sequential piece of memory. We refer to the first element as the index 0
  - The index in computing is the distance (number of characters in this example) from the start, not which element in order.
- You can access an individual character with the index operator []:

```
s = "Welcome"
print(s[4])
# Outputs 'r'
```



THE +, *, [:], AND IN OPERATORS

- Consider:

```
s1 = "Welcome"
s2 = "Python"

s3 = s1 + " to " + s2         # s3 contains "Welcome to Python"

s4 = 2*s1                     # s4 constains "WelcomeWelcome"

s5 = s1[3:6]                  # s5 contains "com"

containsW = 'W' in s1         # containsW stores True
containsXYZ = "XYZ" not in s2 # containsX stores True
```

THE +, *, [:], AND IN OPERATORS

- + – is an operator that **concatenates** (joins) two strings and returns the result
- * – is an operator that repeats a string some amount of times and returns the result (called the **repetition operator**)
- `[ : ]` – is an operator that returns a substring of the string, called the **slicing operator**. The slice returned begins at the first index and ends at the second index - 1.
- `in` and `not in` – are **containment operators** returning Boolean values whether a character/string is contained/not contained within a string or not.

NEGATIVE INDEX IN A SLICING OPERATOR

- Consider:

```
s1 = "Welcome"
s2 = s1[-1]  # s2 contains "e"
s3 = [-3:-1] # s3 contains "om"
```

- A negative index counts from the end of the string

FOREACH LOOPS OVER STRINGS

- Looping over the contents of a string is important for various settings.
- Consider the following with `someString = "Hello"`:

  -
    ```
    for ch in someString:   # Recall for is for each element
                            # of a sequence!
      print(ch, end=",")    # Outputs: H,e,l,l,o
    ```

  -
    ```
    for i in range(0, len(someString), 2):
      print(s[i], end=" ") # Outputs: H l o
    ```

COMPARING STRINGS

- Relational operators work on strings as well. Consider:

```
s1 = "green"
s2 = "glow"
s1 == s2 # False
s1 != s2 # True
s1 > s2  # True
s1 >= s2 # True
s1 < s2  # False
s1 <= s2 # False
```

TESTING CHARACTERS IN A STRING

- The following methods can be used on string objects to test character, example:

```
s = "123"
s.isdigit()
# True
```

str	
isalnum(): bool	Return True if all characters in this string are alphanumeric and there is at least one character.
isalpha(): bool	Return True if all characters in this string are alphabetic and there is at least one character.
isdigit(): bool	Return True if this string contains only number characters.
isidentifier(): bool	Return True if this string is a Python identifier.
islower(): bool	Return True if all characters in this string are lowercase letters and there is at least one character.
isupper(): bool	Return True if all characters in this string are uppercase letters and there is at least one character.
isspace(): bool	Return True if this string contains only whitespace characters.

SEARCHING FOR SUBSTRINGS

- The following methods can be used in order to search for substrings. Example:

```
s.endswith("me")
```

str	
endswith(s1: str): bool	Returns True if the string ends with the substring s1.
startswith(s1: str): bool	Returns True if the string starts with the substring s1.
find(s1): int	Returns the lowest index where s1 starts in this string, or -1 if s1 is not found in this string.
rfind(s1): int	Returns the highest index where s1 starts in this string, or -1 if s1 is not found in this string.
count(subtring): int	Returns the number of non-overlapping occurrences of this substring.

CONVERTING STRINGS

- You can perform transformations on strings for convenience.

str	
capitalize(): str	Returns a copy of this string with only the first character capitalized.
lower(): str	Returns a copy of this string with all characters converted to lowercase.
upper(): str	Returns a copy of this string with all characters converted to uppercase.
title(): str	Returns a copy of this string with the first letter capitalized in each word.
swapcase(): str	Returns a copy of this string in which lowercase letters are converted to uppercase and uppercase to lowercase.
replace(old, new): str	Returns a new string that replaces all the occurrence of the old string with a new string.

STRIPPING WHITESPACE CHARACTERS

- Additionally, stripping whitespace

str	
lstrip(): str	Returns a string with the leading whitespace characters removed.
rstrip(): str	Returns a string with the trailing whitespace characters removed.
strip(): str	Returns a string with the starting and trailing whitespace characters removed.

FORMATTING STRINGS

- Additionally, methods to format.

str	
center(width): str	Returns a copy of this string centered in a field of the given width.
ljust(width): str	Returns a string left justified in a field of the given width.
rjust(width): str	Returns a string right justified in a field of the given width.
format(items): str	Formats a string. See Section 3.6.

EXERCISE

- Write a method that determines whether a string is a palindrome (a string that is the same read forward and backward, e.g., racecar).
- Write a test method that allows a user to enter a string and print whether it is a palindrome or not.

EXERCISE

- Write a function that converts a string of hexadecimal (base 16, e.g., "1F") to a decimal number (31 for the example).
- Write a test for your functions.

OPERATOR OVERLOADING

- There is some magic to how strings support things like concatenation. It turns out we can also write these special methods for operators. This is called operator overloading.
- **Operator overloading** allows the programmer to use the built-in operators for user-defined methods.
  - These methods are named in a special way for Python to recognize the association.

OPERATOR OVERLOADING

- The following are special methods in python. Example:

```python
class Addable:
  def __init__(self, x = 0):
     self.__x = x
  def __add__(self, other):
    return Addable(self.__x + other.__x)
  def __str__(self):
    return str(self.__x)

a1 = Addable(3)
a2 = Addable(2)
a3 = a1 + a2
print(a3) # Outputs 5 because of __str__
```

Operator	Method
+	__add__(self, other)
*	__mul__(self, other)
-	__sub__(self, other)
/	__div__(self, other)
%	__mod__(self, other)
<	__lt__(self, other)
<=	__le__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)
>	__gt__(self, other)
>=	__ge__(self, other)
[index]	__getitem__(self, index)
in	__contains__(self, value)
len	__len__(self)
str	__str__(self)

REMINDER

- Read the book! Lots of great examples and extra explanations.
  - Read/answer end of section exercises/checkpoints