3: String operations

What is a string?

- String is a *sequence of characters.*
- In Python, the length is restricted by amount of free memory only
- Strings are *immutable objects*.

What is a string? (2)

- So these are all examples of strings:

"House"
"a"
""
"In computer programming, a *string* is traditionally a sequence of characters, either as a literal constant or as some kind of variable. The latter may allow its elements to be mutated and the length changed, or it may be fixed (after creation)."

Characters in a string

- A string can contain any characters defined in the current **character set**
- In default, Python uses ASCII encoding.
- This can be changed to Unicode (UTF-16), if enhanced support for special characters is needed, see https://docs.python.org/2/howto/unicode.html

Creating string objects

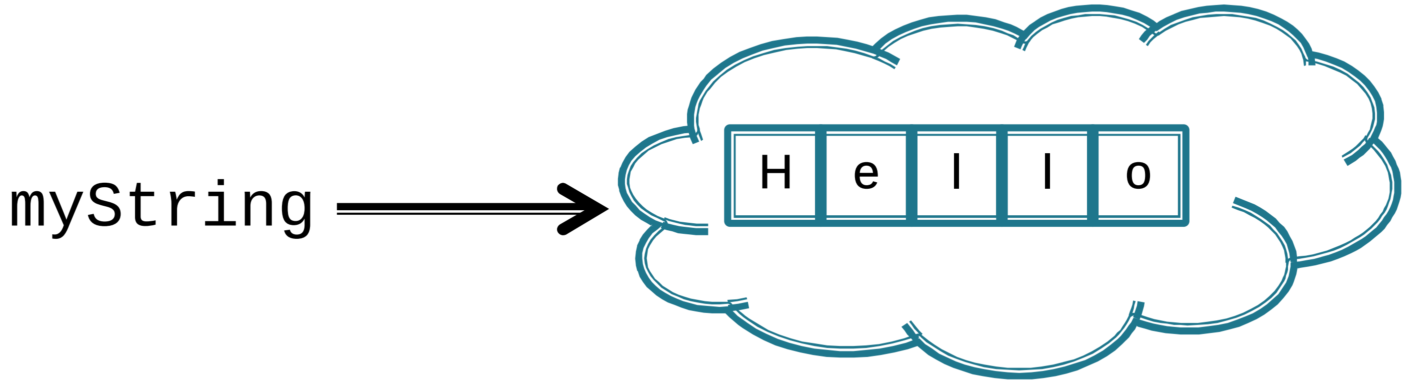
- As mentioned before, a string object is typically created by assigning a **reference to it:**

```
myString = "Hello"
otherString = "Hello" + "!"
print "Hi there"
print "2 + 3 results to " + str(2 + 3)
```

Creating string objects (2)

- Creating a string by assignment creates a string object into memory and stores a reference into variable:

```
    myString = "Hello"
```



Creating string objects (3)

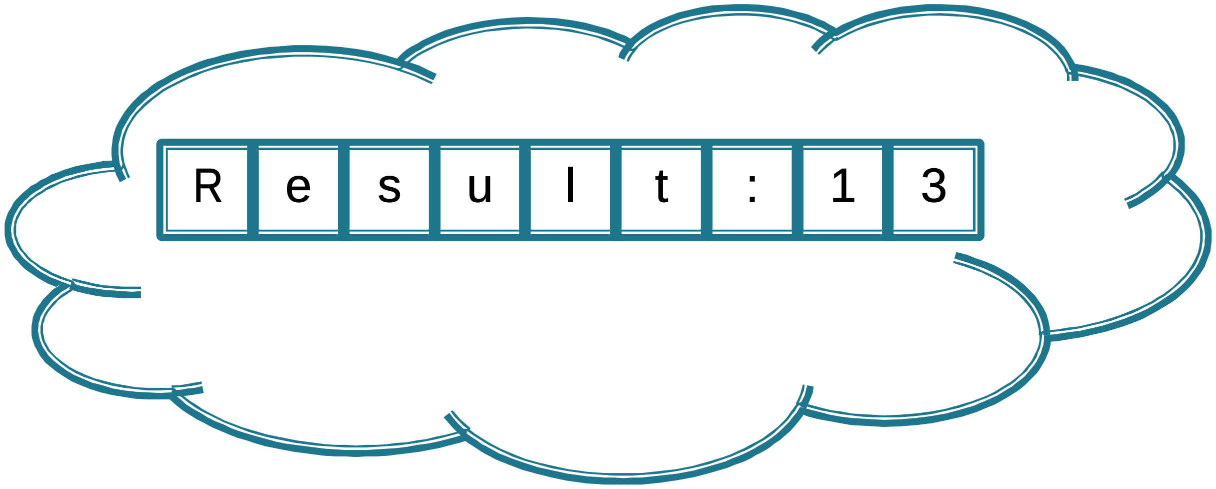
- A string object can be created without constant reference in some cases

- For example using string concatenation in an expression without assignment or in print statement creates a "temporary" string object

- This is similar to any expression with result value not stored

Example

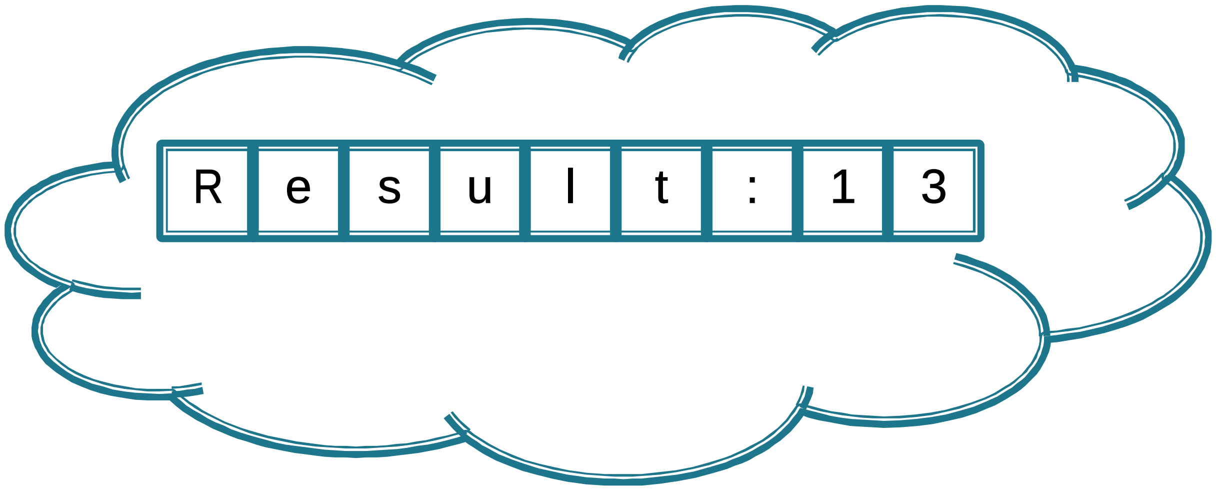
```
print "Result:" + "13"
```



Example

```
print "Result:" + "13"
```

This object can not be accessed later, as no reference is set.



Multi-line strings

- Python supports a creation of multi-line strings by using three quotas:

```
s = """This is a string
containing
more
than one line."""
```

String length

- The length of a string can be returned by using the **len** function.
- The function returns the **length** of a string parameter as an **integer**:

```
print len("abcde") # outputs 5
myVar = "Hello, all!"
print len(myVar) # outputs 11
```

String length (2)

- Hence, the length of a string is the **total number of characters** stored in that string object.

- Whitespace characters (such as space) are also calculated into the length.

String length (3)

- Note, that special characters such as `\n` and `\t` only count as a single character, though they are denoted with two.
- For example, `\n` stands for line feed and...
- ...`\t` for tabulator.

Example

```
myVar = "hello!\n" # Hello and linefeed
print len(myVar) # outputs 7

tabs = "1\t2\t3"
length = len(tabs)
print length # outputs 5
```

Operators usable with strings

Operator	Usage
+	String concatenation
*	String multiplication
[ ]	Extracting a character or a slice from as string

Concatenation

- As seen before, concatenation creates a new string with characters from all concatenated strings:

"abc" + "def" à "abcdef"

Concatenating other objects

- The concatenation only supports strings. This means, that other objects need to be converted into strings with **str** function

```
myVar = "result:" + 1 # throws an error

myVar = "result:" + str(1) # this works
```

Example

- What does the following output?

```
a = 12
b = 10
st = str(a * b)
print len(st)
```

Example 2

- How about this?

```
print 2 + 2 * "ab"
```

indices in strings

- Note, that the indexing of characters in a string always starts from a zero:

0	1	2	3	4	5
H	e	l	l	o	!

- Hence, any string **s** contains characters in indices from 0 to `len(s) – 1`.

Extracting a character from string

- The [ ] –operator can be used to extract a single character from a string.

- The operator gets the character index (an integer) as a parameter:

```
print "abdc"[0] # outputs a
myVar = "hello!"
print myVar[4] # outputs o
```

Extracting a char.. (cont)

- Since the indexing starts at zero, the last character in a string can be output like this:

```
s = "abcdefghi"
print s[len(s) – 1]
```

- ...or in Python, like this:

```
s = "abcdefghi"
print s[-1]
```

Negative indices

- In fact, using negative integer as an index starts indexing in reverse direction:

-6	-5	-4	-3	-2	-1
H	e	l	l	o	!

Extracting a slice from a string

- A slice, containing 1 or more characters, can be also extracted with the [ ] operator.

- The syntax is:

```
    string[startIndex : endIndex]
```

Extracting a slice.. (2)

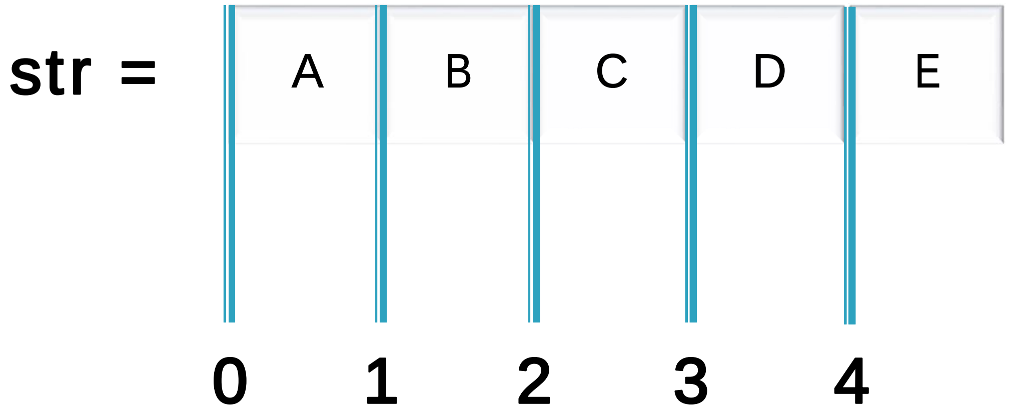
- Note, that the start index is *inclusive,* but the end index *exclusive*.

- Hence, `myString[n:m]` would return all characters between indices
  [n, n+1, n+2, ... , m-2, m-1]

à Thus, the length of the substring (or a **slice**) is end index – start index.

How to remember:

- A good way to remember how end and start indices work is to imagine the indices at the left hand side of the character:



```
str[1:3] à BC
str[0:4] à ABCD
```

Omitting start or end index

- Either of the indices may be omitted. If the start is omitted, it will be replaced with 0; if the end is omitted, it will be replaced with len(string)

- This is often handy when a slice from beginning or end of the string is needed.

Examples

```
myString = "hello all"

print myString[2:] # à llo all

print myString[:3] # à hel

print myString[3:7] # à lo a

print myString[0:11] # à ?
```

Negative indices (2)

- Again, negative indices may be used, but there's a catch:

```
myString = "hello"
print myString[-1:-3]
```

- What does the program above output?

Negative indices (3)

- The program does not output anything, since slice operation starts from the **startIndex** and moves forward one character at a time until the **endIndex** is reached.

- Hence, the length of slice

```
        s[-1:-3]
```

- ...is -3 – (-1) == -2, which in practice becomes zero.

Reverse slices

- For this, the direction of the slice can be set with a third parameter

  string[startIndex : endIndex : step]

- By using -1 as a step, we can define a reverse slice.

Examples

```
myString = "hello"

s = myString[0:2:-1] # "eh"

s = myString[-1:-3:-1] # ol

s = myString[::-1] # "olleh"
```

Examples (2)

- The step can be any integer > 1:

```
s = "abcdefghijklmnopq”
print s[::2] # acegikmoq
print s[::3] # adgjmp
print s[-1:-1:-3] # qnkheb
```

String manipulation

- Very common in programs
- Three subcategories:
  - Counting occurrences
  - Finding occurrences
  - Replacing occurrences

First: methods

- A method in Python is a function that is utilized **via an object**.

- Syntax:

```
objectVariable.methodName(parameters)
```

Counting occurrences

- The **occurrences of a substring** can be counted with **`count()`** *method*.

- The method returns the number of times the given parameter is found in a string.

- Syntax:

```
string.count(substring)
```

Counting occurrences (cont.)

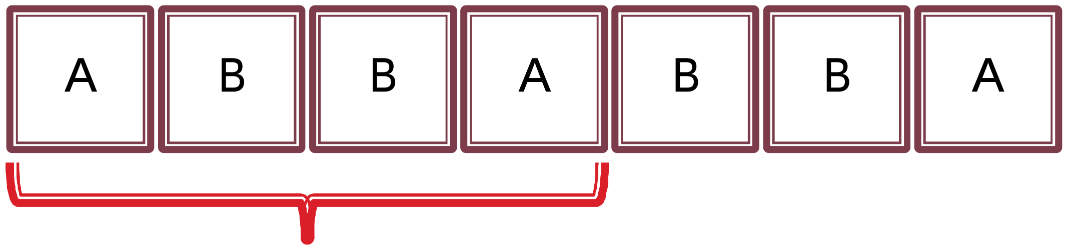
- Example:

```
myString = "abbabbaba"
c = myString.count("bb")
print c # outputs 2

print "this is a string".count("i") # 3

print "ab aba abab".count("ab ") # 1
```

Counting occurrences (cont.)

- Note, that only *the complete occurrences* are counted. Hence, a part of a substring can not be contained in another substring:

myString =



```
print myString.count("ABBA")
#...outputs 1
```

Finding occurrences

- The **first** index of a substring in a string can be returned by using the **`find()`** method.

- The method has a substring as a parameter, and returns the index where the substring first occurs in the original string.

- The method **returns `-1`**, if no occurrence of the substring can be found in the string.

Examples

```
print "abcabc".find("ab") # output 0

print "abcabc".find("bca") # output 1

mystr = "My red house"
loc = mystr.find("house")
print loc # output 7

print "abcabc".find("cb") # output -1
```

Replacing substrings

- To replace substrings in a string, we can use the `replace()` function in Python.
- The method creates and returns a **new string**, where **all occurences** of the first substring are replaced with the second substring.
- Syntax:

```
myString.replace(oldString,newString)
```

Examples

```
myString = "A little green house"
print myString.replace("little","big")

print "abcabc".replace("a","ab")

print "abababab".replace("b","bcd")
```

Replacing substrings (cont.)

- Note, that the method returns **a new string** with substrings replaced. The original string is left intact.

- Hence, to change a string, we need to use something like

```
myString = myString.replace("aa","bb")
```

- ...or get the result in another string:

```
otherString = firstString.replace("hey","hello!")
```

Replacing substrings (cont.)

- Hence, consider following program:

```
st = "Hi there!"
st.replace("Hi", "Hello")
print st
```

- What does the program output?

Strings are immutable

- A string object is immutable. This means, that the content of created string object can **never** change

- Instead, a new string based on existing object can be created.

Strings are immutable (2)

- But what about the next program?

```
st = "Hello all!"
print st

st = "Hi all!"
print st

st = st.replace("Hi", "Hey")
print st
```

Strings are immutable (3)

- In previous example, the **content of the strings** does not change
- Instead, new string objects are created
- In the final case, the new objects are based on existing ones

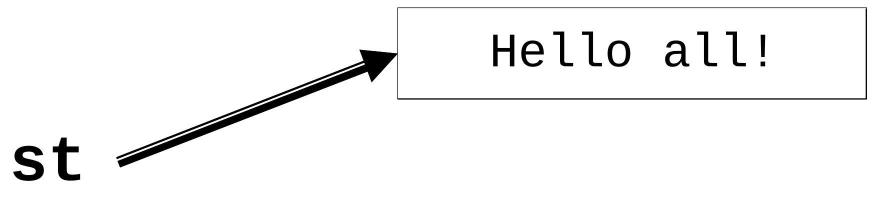
Object vs. variable

- Hence, it is important to distinguish the **object** and the **variable referencing the object.**

- Even if the object is immutable, the value of the **variable** referencing it **can** change

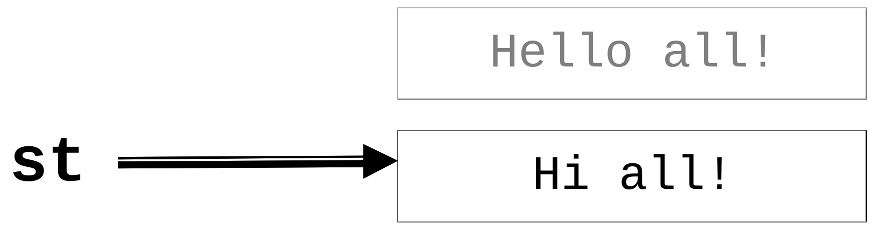
Visualization

```
st = "Hello all!"
st = "Hi all!"
st = st.replace("Hi",
"Hey")
```



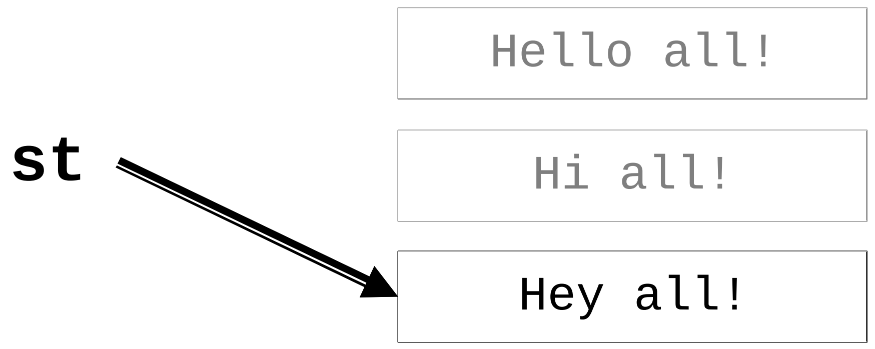
Visualization

```
st = "Hello all!"
st = "Hi all!"
st = st.replace("Hi", "Hey")
```



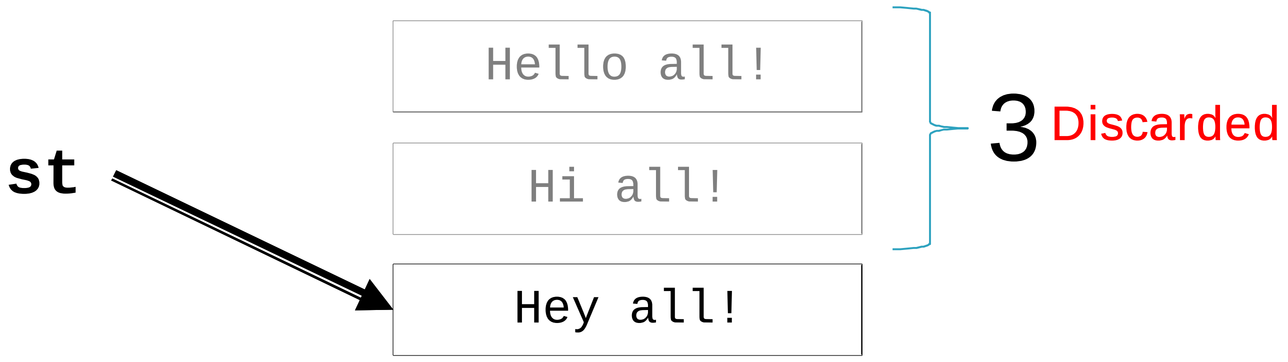
Visualization

```
st = "Hello all!"
st = "Hi all!"
st = st.replace("Hi", "Hey")
```



Visualization

```
st = "Hello all!"
st = "Hi all!"
st = st.replace("Hi", "Hey")
```



Replacing substrings (cont.)

- Consider the following programs:

```
st = "Hi there!"
st.replace("Hi", "Hello")
print st
```

```
st = "Hi there!"
st = st.replace("Hi", "Hello")
print st
```