Strings

Genome 373

Genomic Informatics

Elhanan Borenstein

```
print “hello, world”

pi = 3.14159
pi = -7.2
yet_another_var = pi + 10
print pi

import math
log10 = math.log(10)

import sys
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print arg1, ":", arg2
```

Programs vs. Interpreter

Writing and running a program:

```
print "hello, world!"
```

```
>python hello.py
hello, world!
```

Running code in the interpreter:

```
>>> print "hello, world!"
hello, world!
```

Strings

- A <u>string</u> type object is a sequence of characters.
- In Python, strings start and end with single <u>or</u> double quotes (they are equivalent but they have to match).

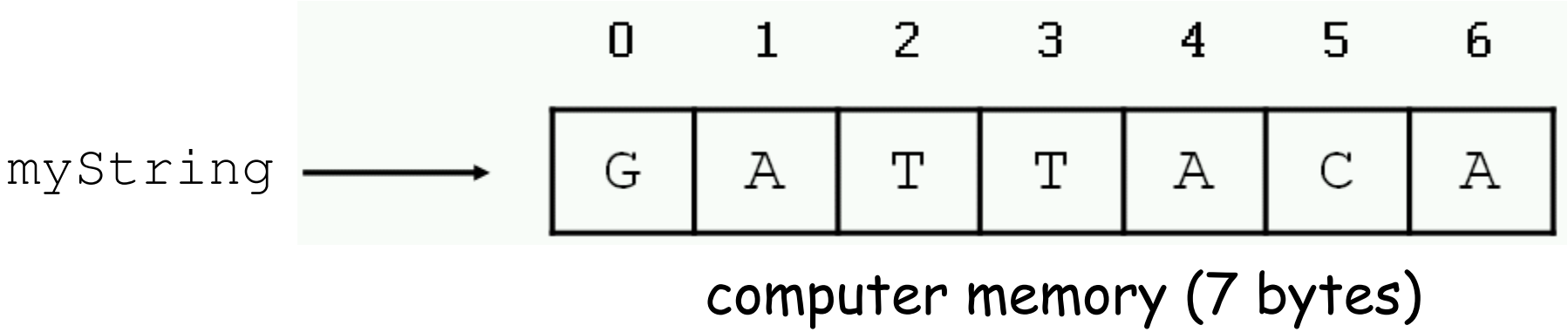
```
>>> s = "foo"
>>> print s
foo
>>> s = 'Foo'
>>> print s
Foo
>>> s = "foo'
SyntaxError: EOL while scanning string literal
```

(EOL means end-of-line; to the Python interpreter there was no closing double quote before the end of line)

Defining strings

- Each string is stored in computer memory as an array of characters.

```
>>> myString = "GATTACA"
```



In effect, the variable `myString` consists of a pointer to the position in computer memory (the address) of the 0th byte above. Every byte in your computer memory has a unique integer address.

How many bytes are needed to store the human genome? (3 billion nucleotides)

Accessing single characters

- You can access individual characters by using indices in square brackets.

```
>>> myString = "GATTACA"
>>> myString[0]
'G'
>>> myString[2]
'T'
>>> myString[-1]
'A'
>>> myString[-2]
'C'
>>> myString[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

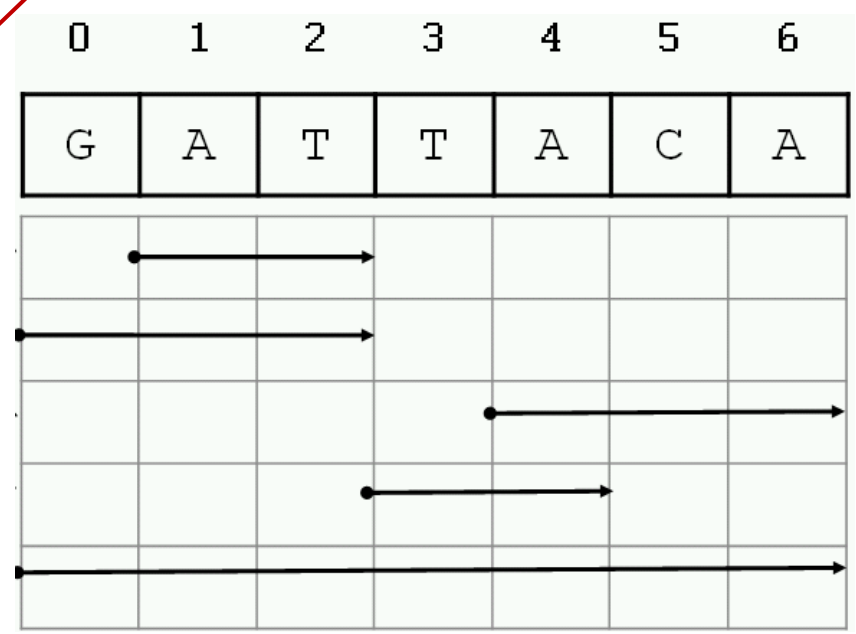
Negative indices start at the end of the string and move left.

FYI - when you request `myString[n]` Python adds `n` to the memory address of the string and returns that byte from memory.

Accessing substrings ("slicing")

```
>>> myString = "GATTACA"
>>> myString[1:3]
'AT'
>>> myString[:3]
'GAT'
>>> myString[4:]
'ACA'
>>> myString[3:5]
'TA'
>>> myString[:]
'GATTACA'
```

shorthand for beginning or end of string



notice that the length of the returned string [x:y] is y - x

Special characters

```
>>> print "He said "Wow!""
SyntaxError: invalid syntax
```

The backslash is used to introduce a special character.

```
>>> print "He said \"Wow!\""
He said "Wow!"

>>> print "He said:\nWow!"
He said:
Wow!
```

Escape sequence	Meaning
\'	Single quote
\"	Double quote
\n	Newline
\t	Tab
\\	Backslash

More string functionality

```
>>> len("GATTACA")
7
>>> print "GAT" + "TACA"
GATTACA
>>> print "A" * 10
AAAAAAAAAA
>>> "GAT" in "GATTACA"
True
>>> "AGT" in "GATTACA"
False
>>> temp = "GATTACA"
>>> temp2 = temp[1:4]
>>> print temp2
ATT
>>> print temp
GATTACA
```

← Length

← Concatenation

← Repeat

(you can read this as "is GAT in GATTACA ?")

← Substring tests

← Assign a string slice to a variable name

String methods

- In Python, a <u>method</u> is a **function** that is defined with respect to a particular object.
- The syntax is:
  `object.method(arguments)`

  or `object.method()` - no arguments

```
>>> dna = "ACGT"
>>> dna.find("T")
3
```

the first position where "T" appears

object (in this case a string object)

string method

method argument

String methods

```
>>> s = "GATTACA"
>>> s.find("ATT")
1
>>> s.count("T")
2
>>> s.lower()
'gattaca'
>>> s.upper()
'GATTACA'
>>> s.replace("G", "U")
'UATTACA'
>>> s.replace("C", "U")
'GATTAUA'
>>> s.replace("AT", "**")
'G**TACA'
>>> s.startswith("G")
True
>>> s.startswith("g")
False
```

Function with no arguments

Function with two arguments

Strings are immutable

- Strings cannot be modified; instead, create a new string from the old one using assignment.

```
>>> s = "GATTACA"
>>> s[0] = "R"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support item
   assignment

>>> w = "R" + s[1:]
>>> print w
RATTACA
>>> print s
GATTACA

>>> s = “R” + s[1:] # THIS WILL WORK!
>>> print s
RATTACA
```

Strings are immutable

- String methods do not modify the string; they <u>return a new string</u>.

```
>>> seq = "ACGT"
>>> print seq.replace("A", "G")
GCGT

>>> print seq
ACGT

>>> new_seq = seq.replace("A", "G")
>>> print new_seq
GCGT
>>> print seq
ACGT
```

assign the result from the right to a variable name

String summary

Basic string operations:

```
S = "AATTGG"          # literal assignment - or use single quotes ' '
s1 + s2               # concatenate
S * 3                 # repeat string
S[i]                  # get character at position 'i'
S[x:y]                # get a substring
len(S)                # get length of string
int(S)                # turn a string into an integer
float(S)              # turn a string into a floating point number
```

Methods:

```
S.upper()
S.lower()
S.count(substring)
S.replace(old,new)
S.find(substring)
S.startswith(substring)
S.endswith(substring)
```

is a special character – everything after it is a comment, which the program will ignore – USE LIBERALLY!!

Printing:

```
print var1,var2,var3          # print multiple variables
print "text",var1,"text"      # print a combination of text and vars
```

Class problem #1

- Write a program called **`dna2rna.py`** that reads a DNA sequence from the first command line argument and prints it as an RNA sequence. Make sure it retains the case of the input.

```
> python dna2rna.py  ACTCAGT
ACUCAGU
> python dna2rna.py actcagt
acucagu
> python dna2rna.py ACTCagt
ACUCagu
```

Two solutions

```
import sys
seq = sys.argv[1]
new_seq = seq.replace("T", "U")
newer_seq = new_seq.replace("t", "u")
print newer_seq
```

OR

```
import sys
print sys.argv[1]
```
(to be continued)

Two solutions

```
import sys
seq = sys.argv[1]
new_seq = seq.replace("T", "U")
newer_seq = new_seq.replace("t", "u")
print newer_seq
```

OR

```
import sys
print sys.argv[1].replace("T","U")
```

(to be continued)

Two solutions

```
import sys
seq = sys.argv[1]
new_seq = seq.replace("T", "U")
newer_seq = new_seq.replace("t", "u")
print newer_seq
```

OR

```
import sys
print sys.argv[1].replace("T","U").replace("t","u")
```

- It is legal (but not always desirable) to chain together multiple methods on a single line.
- Think through what the second program does until you understand why it works.

Tips:

Reduce coding errors - get in the habit of always being aware what type of object each of your variables refers to.

Use informative variable names.

Build your program bit by bit and check that it functions at each step by running it.