Jim Thomas

jht@u.washington.edu

Foege S333C

Homework (send to Tanya)

- Posted to class website on Wednesdays
- Due on Wednesday at 5pm
- Submit via email to me: tgranch@uw.edu
  - Name as: Lastname_Firstname_PS#_whatever.ext

  eg: Grancharova_Tanya_PS1_answers.docx
  Grancharova_Tanya_PS1_code.py
- Written exercises
  - MS Word, OpenOffice, or txt format
- Code/result sets
  - Text format ONLY! (for python scripts extension should be .py)

Python review (from quiz section)

- interactive and script mode
  - use interactive mode to check syntax etc.
  - mostly use script mode: write a program, run it
- python code is built with **objects**, each of which has a **type**
- a **variable** is a name for an object
- the assignment operator is '='
  e.g. `myString = "spring is here!"`
- the **import** command adds functionality not available by default
- command-line arguments are accessed through `sys.argv`

Strings

- A string type object is a sequence of characters.
- In Python, string literals start and end with single or double quotes (but they have to match).

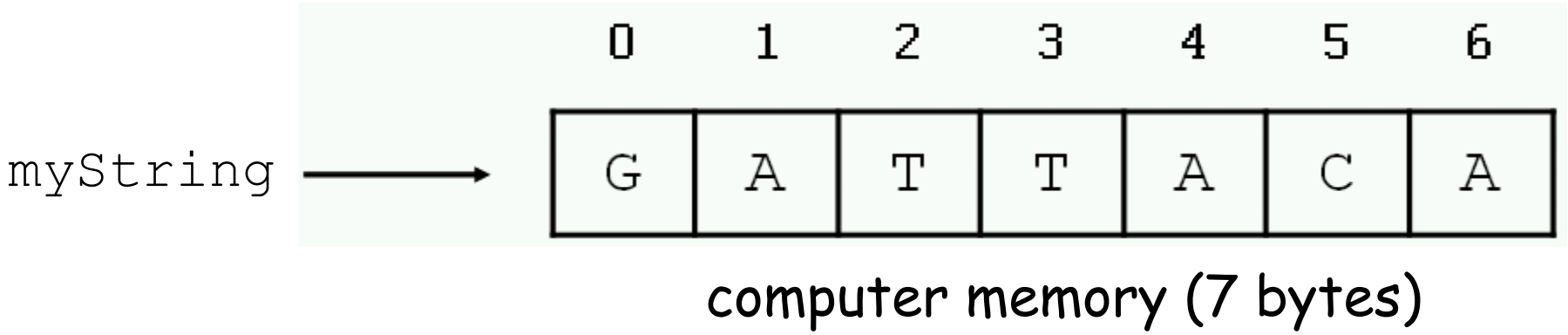
```
>>> s = "foo"
>>> print s
foo
>>> s = 'Foo'
>>> print s
Foo
>>> s = "foo'
SyntaxError: EOL while scanning string literal
```

(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 in sequential bytes.

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



The variable `myString` includes a pointer to the position in computer memory (the address) of the 0th byte above. Every byte in computer memory has a unique address.

Accessing single characters

- 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
```

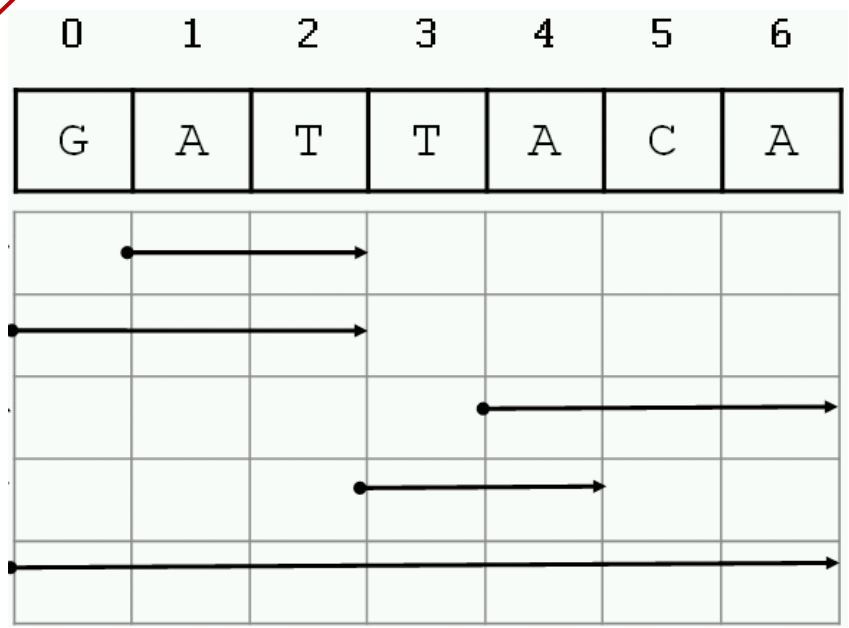
Negative indices start at the end of the string and move left.

When you request `myString[n]` Python adds `n` to the memory address of the string and returns that byte from memory (fast).

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

- The backslash is used to introduce a special character.

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

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

whenever Python runs into a backslash in a string it interprets the next character specially

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 for a particular type of 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

Some of many 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, comma separated

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
>>> s = "R" + s[1:]
>>> print s
RATTACA
>>> s = s.replace("T","B")
>>> print s
RABBACA
>>> s = s.replace("ACA", "I")
>>> print s
RABBI
>>> s
'RABBI'
```

Try to change the zeroth character - illegal

print the string

the string itself (type shown by the single quotes)

Strings are immutable

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

```
>>> seq = "ACGT"
>>> 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

(also see Python quick reference guide linked from course web page)

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 decimal 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 literal text (strings) and variables
```

Numbers

- Python defines various types of numbers:
  - Integer (1234)
  - Floating point number (12.34)
  - Octal and hexadecimal number (0177, 0x9gff)
  - Complex number (3.0+4.1j)
- You will likely only use the first two.

Conversions

```
>>> 6/2
3
>>> 3.0/4.0
0.75
>>> 3/4.0
0.75
>>> 3*4.0
12.0
>>> 3*4
12
>>> 3/4
0
```

integer → float

- The result of a mathematical operation on two numbers of the <u>same</u> type is a number of that type.
- The result of an operation on two numbers of <u>different</u> types is a number of the more complex type.

watch out - integer divisions are truncated rather than rounded

Formatting numbers

- The % operator formats a number.
- The syntax is `<format> % <number>`
- `format` is a string

```
>>> "%f" % 3    # print as float
'3.000000'
>>> "%.2f" % 3 # print as float with
'3.00'          # 2 digits after decimal
>>> "%5.2f" % 3 # width 5 characters
' 3.00'
```

Formatting codes

- `%i` = integer (or %d)
- `%f` = float value (decimal number)
- `%e` = scientific notation
- `%g` = general, easily readable notation (uses decimal notation unless there are too many zeroes, then switches to scientific notation)

More complex formats

%[flags][width][.precision][code]

Left justify ("-")
Include numeric sign ("+")
Fill in with zeroes ("0")

Total width of output

Number of digits after decimal

i, f, e, g

Examples (review later)

```
>>> x = 7718
>>> "%i" % x
'7718'
>>> "%-6i" % x
'7718  '
>>> "%06i" % x
'007718'
>>> x = 1.23456789
>>> "%i" % x
'1'
>>> "%f" % x
'1.234568'
>>> "%e" % x
'1.234568e+00'
>>> "%g" % x
'1.23457'
>>> "%g" % (x * 10000000)
'1.23457e+07'
```

Read as "use the preceding code to format the following number"

Don't worry if this all looks like Greek – you can figure out how to do these when you need them in your programs. After a while they are pretty easy.

Ιτ συρε λοοκσ λικε Γρεεκ το με.

(It sure looks like to Greek to me)

Sample 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
```

Hint: first get it working for uppercase letters and then extend it to lowercase and mixed case.

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
```

```
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
```

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

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

Sample problem #2

- Write a program get-codons.py that reads the first command line argument as a DNA sequence and prints the first three codons, one per line, in uppercase letters.

```
> python get-codons.py TTGCAGTCG
TTG
CAG
TCG
> python get-codons.py TTGCAGTCGATCTGATC
TTG
CAG
TCG
> python get-codons.py tcgatcgactg
TCG
ATC
GAC
```

(slight challenge – print the codons on <u>one line</u> separated by spaces)

Solution #2

```
# program to print the first 3 codons from a DNA
# sequence given as the first command-line argument
import sys
seq = sys.argv[1]  # get first argument
up_seq = seq.upper()  # convert to upper case
print up_seq[0:3]  # print first 3 characters
print up_seq[3:6]  # print next 3
print up_seq[6:9]  # print next 3
```

These comments are simple, but when you write more complex programs good comments will make a huge difference in making your code understandable (both to you and others).

Sample problem #3

- Write a program that reads a protein sequence as a command line argument and prints the location of the first cysteine residue (C).

```
> python find-cysteine.py
MNDLSGKTVIITGGARGLGAEAARQAVAAGARVVLADVLDEEGAATARELGDAARYQHLDVTI
EEDWQRVCAYAREEFGSVDGL
70
> python find-cysteine.py
MNDLSGKTVIITGGARGLGAEAARQAVAAGARVVLADVLDEEGAATARELGDAARYQHLDVTI
EEDWQRVVAYAREEFGSVDGL
-1
```

note: the `-1` here means that no C residue was found

Solution #3

```
import sys
protein = sys.argv[1]
upper_protein = protein.upper()
print upper_protein.find("C")
```

(Always be aware of upper and lower case for sequences - it is valid to write them in either case. This is handled above by converting to uppercase so that 'C' and 'c' will both match.)

Challenge problem

- Write a program get-codons2.py that reads the first command- line argument as a DNA sequence and the second argument as the frame, then prints the first three codons in that frame on one line separated by spaces.

```
> python get-codons2.py TTGCAGTCGAG 0
TTG CAG TCG
> python get-codons2.py TTGCAGTCGAG 1
TGC AGT CGA
> python get-codons2.py TTGCAGTCGAG 2
GCA GTC GAG
```

Challenge solution

```
import sys
seq = sys.argv[1]
frame = int(sys.argv[2])
seq = seq.upper()
c1 = seq[frame:frame+3]
c2 = seq[frame+3:frame+6]
c2 = seq[frame+6:frame+9]
print c1, c2, c3
```