Strings

Genome 559: Introduction to Statistical and Computational Genomics

Prof. James H. Thomas

Review

Run a program by typing at a terminal (command) prompt.

Type `python` (enter) at the terminal prompt to enter the Python IDLE interpreter. Prompt changes to `>>>`. Ctrl-D or exit() to quit IDLE.

`python myprog.py` (enter) at the terminal prompt will run the program `myprog.py` in the present working directory.

`python myprog.py arg1 arg2` (etc.) will provide command line arguments arg1 and arg2 (etc.) to the program.

Each argument is a string object - access using `sys.argv[0]`, `sys.argv[1]`, etc., where the program name is the zeroth element.

Write your program with a text editor and save it in the present working directory before running it.

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).

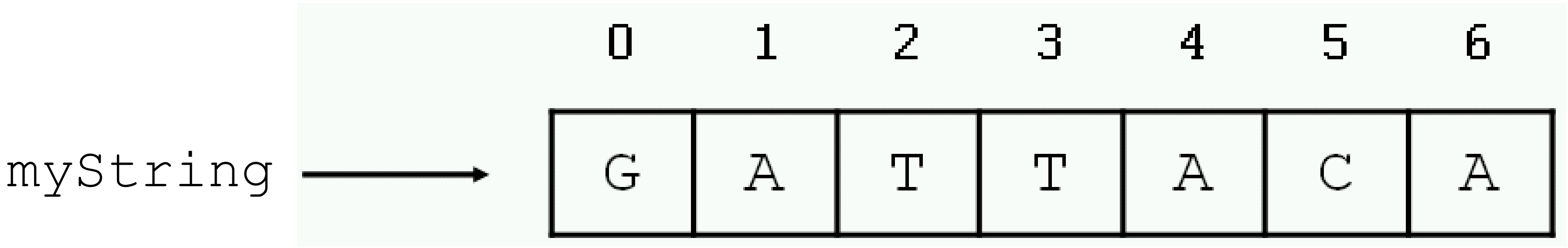
```
>>> s = "foo"
>>> print s
foo
>>> s = 'Foo'
>>> print s
Foo
>>> s = "foo'
SyntaxError: EOL while scanning string literal
```

(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"
```



computer memory (7 bytes)

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

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

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
```

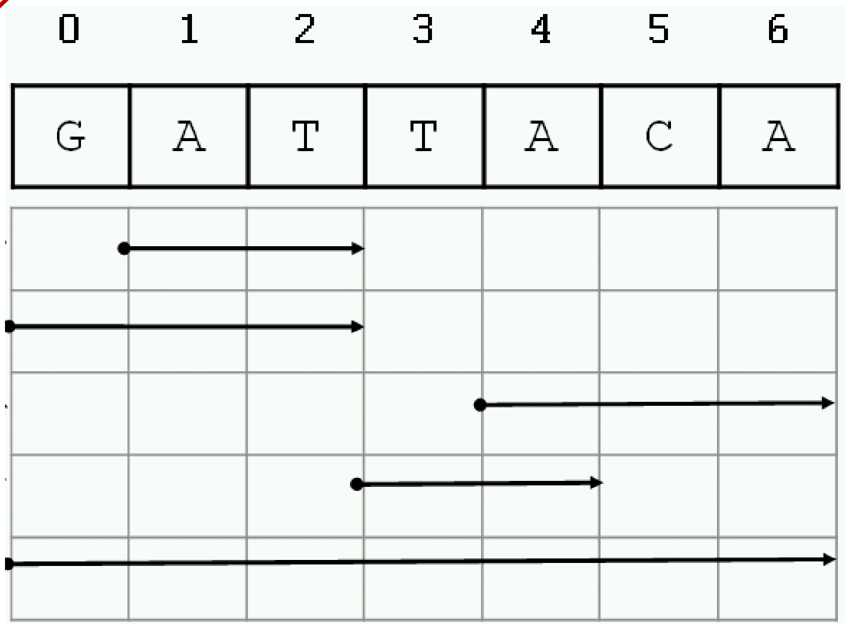
Negative indices start at the end of the string and move left.

FYI - 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



0	1	2	3	4	5	6
G	A	T	T	A	C	A

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!
```

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

Escape sequence	Meaning
\\	Backslash
\’	Single quote
\”	Double quote
\n	Newline
\t	Tab

More string functionality

```
>>> len("GATTACA")
7
```

←Length

```
>>> print "GAT" + "TACA"
GATTACA
```

←Concatenation

```
>>> print "A" * 10
AAAAAAAAAA
```

←Repeat

```
>>> "GAT" in "GATTACA"
```

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

```
True
>>> "AGT" in "GATTACA"
False
```

←Substring tests

```
>>> temp = "GATTACA"
>>> temp2 = temp[1:4]
>>> print temp2
ATT
>>> print temp
GATTACA
```

← Assign a string slice to a variable name

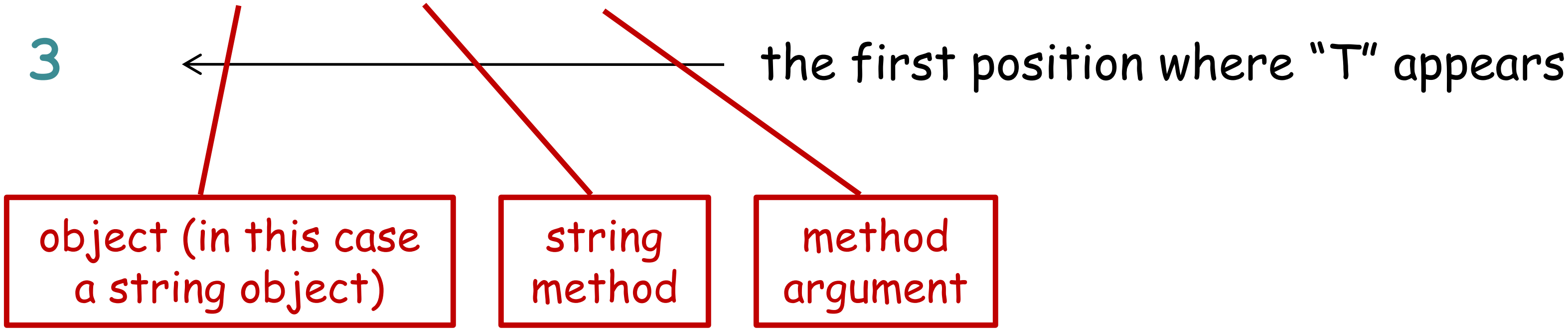
String methods

- In Python, a method 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
```

Method with no arguments

Method with two arguments, comma separated

Strings are immutable

- Strings cannot be modified; instead, create a new string 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 content

the string object (type shown by the quotes)

Strings are immutable

- String methods do not modify the string; they return a new string.

```
>>> 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)
```

Printing:

```
print var1,var2,var3      # print multiple variables
print "text",var1,"text"  # print a combination of literal text (strings) and variables
```

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

Coding Tips:

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

Use informative variable names. (At the start, even including the type in the name is not a bad idea: arg1str, arg1int, mylist1, etc.)

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

Ending a sentence with a preposition is something up with which I will not put. – Winston Churchill

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
# assign argument, replace characters, print
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 chain 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 one line 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
```

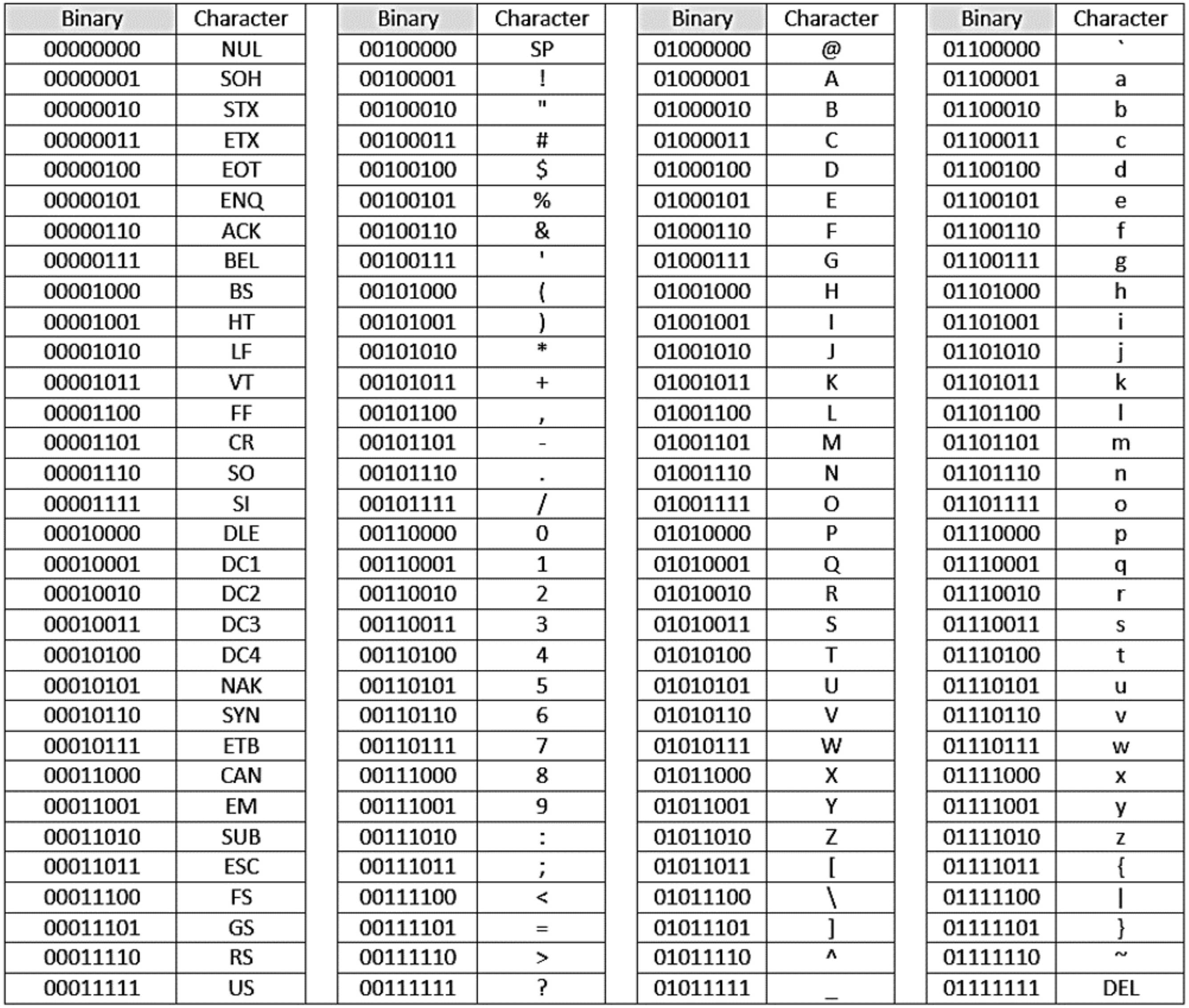
Challenge solution

```
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
```

Reading

- Chapters 2 and 8 of *Think Python* by Downey.

The first 128 ASCII characters (of 256 = 1 byte = 8 bits = 2^8)



Binary	Character	Binary	Character	Binary	Character	Binary	Character
00000000	NUL	00100000	SP	01000000	@	01100000	`
00000001	SOH	00100001	!	01000001	A	01100001	a
00000010	STX	00100010	"	01000010	B	01100010	b
00000011	ETX	00100011	#	01000011	C	01100011	c
00000100	EOT	00100100	$	01000100	D	01100100	d
00000101	ENQ	00100101	%	01000101	E	01100101	e
00000110	ACK	00100110	&	01000110	F	01100110	f
00000111	BEL	00100111	'	01000111	G	01100111	g
00001000	BS	00101000	(	01001000	H	01101000	h
00001001	HT	00101001	)	01001001	I	01101001	i
00001010	LF	00101010	*	01001010	J	01101010	j
00001011	VT	00101011	+	01001011	K	01101011	k
00001100	FF	00101100	,	01001100	L	01101100	l
00001101	CR	00101101	-	01001101	M	01101101	m
00001110	SO	00101110	.	01001110	N	01101110	n
00001111	SI	00101111	/	01001111	O	01101111	o
00010000	DLE	00110000	0	01010000	P	01110000	p
00010001	DC1	00110001	1	01010001	Q	01110001	q
00010010	DC2	00110010	2	01010010	R	01110010	r
00010011	DC3	00110011	3	01010011	S	01110011	s
00010100	DC4	00110100	4	01010100	T	01110100	t
00010101	NAK	00110101	5	01010101	U	01110101	u
00010110	SYN	00110110	6	01010110	V	01110110	v
00010111	ETB	00110111	7	01010111	W	01110111	w
00011000	CAN	00111000	8	01011000	X	01111000	x
00011001	EM	00111001	9	01011001	Y	01111001	y
00011010	SUB	00111010	:	01011010	Z	01111010	z
00011011	ESC	00111011	;	01011011	[	01111011	{
00011100	FS	00111100	<	01011100	\	01111100	|
00011101	GS	00111101	=	01011101	]	01111101	}
00011110	RS	00111110	>	01011110	^	01111110	~
00011111	US	00111111	?	01011111	_	01111111	DEL

Some of the “characters” are written out, e.g. SP is the space character