Computer Programming I

Strings, Lists, Tuples, Sets, and Dictionaries

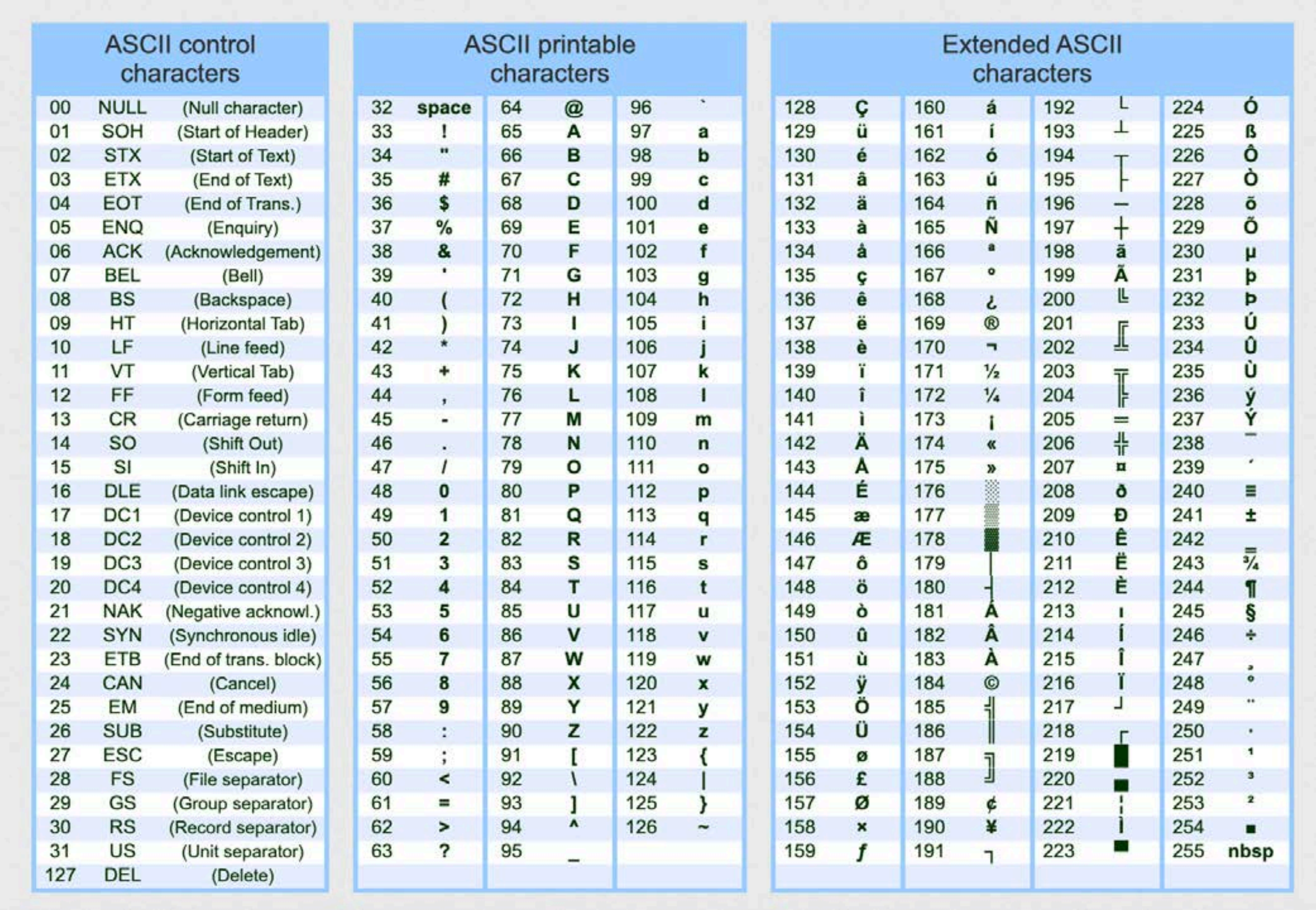
Part 1:

Strings

Representing text

- In programming string variables represent text
- Text is enclosed in quotes (single or double)
- Example: `'hi'  'python'  "A" "programming"`
- A string is a sequence of characters
- Python uses Unicode to represent each possible character as a unique number.
  - Unicode is a character encoding standard
  - http://unicode.org/charts/
  - http://www.asciitable.com/

ASCII Table



ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Empty string

- An empty string is a string that does not contain any characters.
- An empty string is creating by using two quotes, one immediately after the other, nothing in between.
- Example:

```
    empty = ""
    empty2  = ''
```

The length of a string

- We can get the length of a string (how many characters are in the string) with the **`len()`** function.

```
>>> text = "hello there!"
>>> len(text)
>>> 12
```

- The **`len()`** function counts **every** character in the string, including letters, digits, symbols, white spaces, etc.

String as a sequence

- A string is a sequence of characters
- Example: `myStr = 'Hello'`

H	e	l	l	o

- In programing we assign each character in this sequence of characters an index (a.k.a. subscript)
- The index system allows us to reach any character, at any time, without having to go through every character in the string.

The index system

- The index of the first character is always zero
- Index values are whole numbers, and increase by one
- The index of the last character is always the number of characters in the string minus 1

H	e	l	l	o
0	1	2	3	4

- This string has 5 characters
- First character H at index 0
- Last character o at index 4

Getting parts of strings

- We use the subscript operator `[]` to get parts of strings.
- `string[n]` gives you the nth character in the string
  - Only use integer values

Examples

```
>>> phrase = "welcome to Python"
>>> print(phrase[0])
w
>>> print(phrase[5])
m
>>> print(phrase[11])
P
>>> print(phrase[7])

>>>
```

Getting parts of strings

- Negative numbers can be used to access characters from the rightmost character of the string
- Be careful not to use an index value beyond the last index
- Examples:

```
>>> phrase = 'welcome to Python'
>>> print(phrase[-1])
n
>>> print(phrase[-6])
P
>>>
```

Slice Notation

- Slice Notation allows you to select parts of a String
- Format: `aString[start : end : step]`
- Common Usages:
  - Select a Substring `aString [start : end]`
  - Skip Character `aString [start : end : step]`
  - Reverse Chars `aString [end : start : -1]`
  - Select till end `aString [start : ]`
  - Start from beginning `aString [ : end]`

Slice Notation – Example

```
>>> alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> print(alphabet[0:10])
ABCDEFGHIJ
>>> print(alphabet[0:10:2])
ACEGI
>>> print(alphabet[10:2:-1])
KJIHGFED
>>> print(alphabet[10:])
KLMNOPQRSTUVWXYZ
>>> print(alphabet[:7])
ABCDEFG
>>> print(alphabet[:])
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

String Concatenation

- The **+** symbol can be used to append one string at the end of another
  - This operation is known as concatenation
- Examples:

```
>>> print ("hello"  + "Joe")
helloJoe
>>> print ("hello"  + " " + "Joe”)
hello Joe
>>> word1 = "cat"
>>> word2 = "fish"
>>> word3 = word1 + word2
>>> print (word3)
catfish
```

Multiplying Strings

- The * symbol can be used to append multiple copies of one string at the end of itself
- Examples:

```
>>> print ("$" * 10)
$$$$$$$$$$
>>> dollar = "$"
>>> dollarTen = dollar * 10
>>> print (dollarTen)
$$$$$$$$$$
```

Type conversions

- Implicit type conversions occur when an arithmetic expression mixes integers and floats
- Examples:

```
>>> print (5 + 5.5)
10.5
>>> print (5 + 5.0)
10.0
>>> print (10 - 2.5)
7.5
>>> print (10 * 2.0)
20.0
```

Type conversions

- Explicit type conversions can be performed by the programmer by using the functions `int()`, `float()`, and `str()`

The **int()** function

- The **int()** function can be used to convert a string into an integer number
- Example:

```
ticket = "101"
number  = int(ticket)  #number = 101
```

- The string must contain only digits

```
>>> print ( int ("1.2") )
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.2‘
```

- The **int()** function can also be used to force a float into an integer

```
aFloat = 5.25
number  = int(aFloat)  #number = 5
```

The `float()` function

- The `float()` function can be used to convert a string into a floating-point number
- Example:

```
 average = "10.56"
 number = float(average)  #number = 10.56
```

- The `float()` function can also be used force an integer value into a float
- Example:

```
n = float (101)  #n becomes 101.0
```

The str() function

- The str() function converts a number into a string
- Example:

```
num = 99
num2 = 10.55
strNum = str(num)    #strNum is "99"
strNum2 = str(num2)  #strNum2 is "10.55"
```

ord() and chr()

- The **ord** function receives a character and it returns the equivalent Unicode integer for that character

```
>>> ord("A")
>>> 65
```

- The **chr** function receives a number and it returns the character encoded with that number in Unicode

```
>>> chr(65)
>>> "A"
```

Escape Sequences

- Special commands that allow you to control the way the output is displayed
- Formed by a backslash (\ ) followed by another character

Escape Sequences

Sequence	Name	Description
\n	Newline	Causes the cursor to go to the next line
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop
\a	Alarm	Causes the computer to beep
\b	Backspace	Causes the cursor to back up, or move left one position
\r	Return	Causes the cursor to go to the beginning of the current line (or the next line - machine dependent)
\\	Backslash	Causes a backslash to be printed
\’	Single quote	Causes a single quotation mark to be printed
\"	Double quote	Causes a double quotation mark to be printed

Escape Sequences - Examples

```
>>> print(‘Hello\twelcome to \nCS 171')
Hello       welcome to
CS 171

>>> print('Path = C:\\CS171\\Examples\\week2.py')
Path = C:\CS171\Examples\week2.py

>>> print ("M. Ali once said: \"Don't count the days,
            make the days count\"")
M. Ali once said: "Don't count the days, make the
days count"
```

String Formatting

- String Formatting can be used to insert data into strings.
- Placeholders are put inside the string.
- Data is added in the current format.
- Conversion Specifiers:

%d	Substitute integer
%f	Substitute Floating Point
%s	Substitute String
%x	Substitute as Hexadecimal
%e	Substitute as Scientific Notation

Printing with formatting

- Format:

```
(Formatted String) % (variables to subs)
```

- Each substitution position starts with a %
- The data type and format options follow the %
- Example:

```
>>> number = 12.9832
>>> print ("The cost is %f" %number )
The cost is 12.983200
```

Sorting Data into columns

- A column width can be given for the format.
- This lets you align data.
- Put the column width between % and the conversion specifier.

Example

```
number1 = 25
number2 = 391
number3 = 9000
print("Row  | Value ”)
print("%4d | %4d" % ( 1 , number1 ))
print("%4d | %4d" % ( 2 , number2 ))
print("%4d | %4d" % ( 3 , number3 ))
```

Output:

```
Row   | Value
   1 |   25
   2 |  391
   3 | 9000
```

Float Precision

- Floating Point Numbers can be given a precision
- Put a decimal point and the number of digits in between % and the conversion specifier.
- Example:

```
price = 125.678
print("The cost is $%.2f" %price)
```

Output

```
The cost is $125.68
```

Part 2:

Lists, Tuples, Dictionaries, and Sets

Lists

- Lists contains a collection or sequence of values
- Python can create lists of any type
    - Lists can contain strings, numbers, even other lists
    - List can contain a mix of types
- Each item in the list is called an element
- We can use lists to process a variety of types of data.
- To define a list, use the [ ] and separate the elements with commas.

Examples

```
>>> emptyList = []
>>> names = [ "Ana", "Bob", "Claire", "Dylan" ]
>>> grades = [100, 98, 88, 75]
>>> mixedList = ["Jane", "Bennet", 25, 45000.50]

>>> print(names)
['Ana', 'Bob', 'Claire’, ‘Dylan’]
```

Lists and access to elements

- We use the subscript operator [ ] to access elements in a list
- Use a valid index value
  - An integer value
  - First element is at index zero
  - Index of the last element is the number of elements minus 1
- Negative numbers can be used to access elements from the rightmost element of the list
- Use the colon [:] to get slices of a list

Examples

```
>>> names = [ "Ana", "Bob", "Claire", "Dylan" ]
>>> print (names[0])
Ana
>>> print (names[2])
Claire
>>> print (names[-1])
Dylan
>>> print (names[1:3])
['Bob', 'Claire']
```

More operations with lists

- A list value can be changed just like a variable

```
>>> names = [ "Ana", "Bob", "Claire", "Dylan" ]
>>> names[1] = 'Brett'
>>> print(names)
['Ana', 'Brett', 'Claire', 'Dylan']
```

- You can use the + operator to concatenate two lists

```
>>> first = [10, 20, 30]
>>> second = [40, 50]
>>> third = first + second
>>> print(third)
[10, 20, 30, 40, 50]
```

Some useful list functions

- `max(`*`list`*`)` returns the largest value in the list.
- `min(`*`list`*`)` returns smallest value in the list.
- `sum(list)` returns the sum of all the values in the list (numbers only).
- `len(`*`list`*`)` returns the number of elements in that list (the size of the list) as an integer.

Examples

```
>>> scores = [10.0, 9.5, 10.0, 9.0, 8.5, 10.0]
>>> size = len(scores)
>>> total = sum(scores)
>>> minimum = min(scores)
>>> maximum = max(scores)
>>> print(size, total, minimum, maximum)
6 57.0 8.5 10.0
```

Methods and the dot notation

- All data in Python are actually objects
- Objects combine data and methods that act on the object
- Methods are special functions that only objects of the same type understand.
  - Methods are functions known only to certain objects
- To use a method, you use dot notation

```
object.method()
```

Methods to use with lists

- **`append(element)`** adds `element` at the end of the list.
- **`remove(element)`** removes the first occurrence of `element` from the list, if it's there.
- **`pop(index)`** removes the element at the given `index`.
- **`index(element)`** finds the index of the first occurrence of `element` in the list.
- **`count(element)`** tells you the number of times that `element` appears in the list. It returns an integer.

Examples

```
>>> fruits = ['apples', 'grapes', 'cherries', 'apples', 'pears']
>>> fruits.count('apples')
2
>>> fruits.index('apples')
0
>>> fruits.append('kiwi')
>>> fruits
['apples', 'grapes', 'cherries', 'apples', 'pears', 'kiwi']
>>> fruits.remove('apples')
>>> fruits
['grapes', 'cherries', 'apples', 'pears', 'kiwi']
>>> fruits.pop(1)
'cherries'
>>> fruits
['grapes', 'apples', 'pears', 'kiwi']
```

List Example

```
print('Enter 3 numbers on each line')
numbers = []  #empty list - no items

numbers.append(int(input('Enter the first number: ')))
numbers.append(int(input('Enter the second number: ')))
numbers.append(int(input('Enter the third number: ')))

print('The minimum number was', min(numbers))
print('The maximum number was', max(numbers))
```

List Example

```
>>> %Run listBasic.py
Enter 3 numbers on each line
Enter the first number: 10
Enter the second number: 5
Enter the third number: 8
The minimum number was 5
The maximum number was 10
```

List Generation

- Python allows you to use an expression to generate a list.
- **`range(a, b)`** generates every number between a and b, excluding b.
- Examples:

```
>>> numbers = [ x for x in range(0, 10) ]
>>> print(numbers)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> even = [ x for x in range(0, 10, 2)]
>>> print(even)
[0, 2, 4, 6, 8]

>>> odd = [ x for x in range(1, 10, 2)]
>>> print(odd)
[1, 3, 5, 7, 9]
```

List of Inputs

- We can generate a list of inputs.
- This allows you to quickly get multiple inputs from the user.
- Example:

```
>>> inputs = [input('Enter a number:') for x in range(0, 5)]
Enter a number: 10
Enter a number: 20
Enter a number: 30
Enter a number: 40
Enter a number: 50
>>> print(inputs)
['10', '20', '30', '40', '50']
```

Example: calculate quiz averages

- Problem: write a script that reads 4 grades and computer the average of the grades entered by the user
- Process:
  - Make a list of Quiz Grades
    - Get 4 grades using range(0, 4)
    - Use the input command to get items
  - Compute the average of the list using the sum and len functions
  - Display the result

Example: calculate quiz averages

```
quizzes = [float(input("Enter Quiz %d Grade:"%(x + 1))) for x in range (0, 4)]
average = sum(quizzes)/len(quizzes)
print("Quiz Average: %.2f" %average)
```

```
>>> %Run listBasic.py
Enter Quiz 1 Grade: 98.75
Enter Quiz 2 Grade: 99
Enter Quiz 3 Grade: 100
Enter Quiz 4 Grade: 76.50
Quiz Average: 93.56
```

Tuples

- Tuples are similar to lists
  - Same index system
  - Similar functions are available
- but they are immutable
  - Once a tuple is created, we cannot change its elements
    - Elements cannot be changed, added, or removed.
- To define a tuple, use the ( ) and separate the elements with commas

Examples

```
>>> pair = (3, 5)
>>> pair
(3, 5)
>>> pair[0]
3
>>> pair[1]
5
>>> len(pair)
2
>>> min(pair)
3
>>> pair.count(5)
1
>>> pair.index(3)
0
```

Dictionary

- A container used to describe associative relationships
- Represented by the `dict` object type
- A dictionary maps keys with values
  - Key is a term that can be located in the dictionary
    - Keys are unique- each one can only be used once
    - Could only be: string, tuple, or number
  - Value describe data associated with key
    - Could be any type and can be repeated
- To define a dictionary, use the `{}` to surround `key:value` pairs.
  - Separate `key:value` pairs with commas

Example

```
>>> states = {'New Jersey' : 'NJ', 'Delaware' : 'DE',
              'Pennsylvania' : 'PA'}
>>> print(states)
{'New Jersey': 'NJ', 'Delaware': 'DE', 'Pennsylvania': 'PA'}

>>> empty = { }
>>> print(empty)
{}
```

Dictionary access to elements

- Dictionary entries are not ordered by position
  - No index or subscript
- To access a dictionary entry, use the key inside the `[]`
- Entries in a dictionary can be added, deleted and modified as needed
  - `dictionary[key] = value` adds a new pair if it doesn't exist
  - `dictionary[key] = value` modifies existing entry if it exists
  - `del dictionary[key]` deletes entry if it exists

Example

```
>>> print(states)
{'New Jersey': 'NJ', 'Delaware': 'DE', 'Pennsylvania': 'PA’}

>>> print(states['New Jersey'])
NJ

>>> print(states['New York'])
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
KeyError: 'New York'

>>> states['New York'] = 'NY'
>>> print (states)
{'New Jersey': 'NJ', 'Delaware': 'DE', 'Pennsylvania': 'PA', 'New York':
'NY'}

>>> del states['Delaware']
>>> print (states)
{'New Jersey': 'NJ', 'Pennsylvania': 'PA', 'New York': 'NY'}
```

Example: measurement conversion

- Use a dictionary to perform the conversion as indicated by the following table:

1 cup	1 cup
1 Pint	2 Cups
1 Quart	4 Cups
1 Gallon	16 Cups

Example: measurement conversion

```
# define dictionary
fluid = {"cups" : 1,
         "pints" : 2 ,
         "quarts" : 4 ,
         "gallons" : 16
         }

# get input from user
print("Fluid Conversion")
cups = float(input("Enter a number of cups: "))
print("Units: cups, pints, quarts, or gallons")
units = input ("Enter target units: ")

#perform calculations and output results
result = cups / fluid[units]
print(cups, "cups is", result , units)
```

Example: measurement conversion

```
>>> %Run listBasic.py
Fluid Conversion
Enter a number of cups: 28
Units: cups, pints, quarts, or gallons
Enter target units: gallons
28.0 cups is 1.75 gallons
```

Sets

- A set is an unordered collection of unique elements
- Elements in the set do not have a position or index.
- Elements are unique:
  - No elements in the set share the same value.
- A set can be created using the **`set()`** function, which accepts a sequence-type iterable object (list, tuple, string, etc.)
- A set literal can be written using curly braces **`{ }`** with commas separating set elements.
- Note that an empty set can only be created using **`set()`**

Sets

- Examples

```
>>> mySet = set( [1, 2, 3, 4] )
>>> print(mySet)
{1, 2, 3, 4}
>>> mySet2 = { 5, 6, 7 }
>>> print(mySet2)
{5, 6, 7}
```

- Sets can be used to remove duplicates:

```
>>> myList = [1, 2, 3, 4, 4, 5, 6, 2, 7, 1, 2, 3]
>>> mySet = set(myList)
>>> print(mySet)
{1, 2, 3, 4, 5, 6, 7}
```

Set Operations

`len(set1)`	Number of Elements in Set
`set1.update(set2)`	Add all elements from `set2` into `set1`
`set.add(value)`	Add `value` to set
`set.remove(value)`	Remove `value` from set
`set.pop()`	Remove an arbitrary element from set
`set.clear()`	Clears all elements from set