Strings and Characters in Python

Computers store characters as numbers. The assignment must include more characters than you might expect. Many of them are invisible to humans (for example whitespace ' ' and some control characters, to control I/O devices), but essential to computers. In case you would like to know '\n' is one of the special control characters, this is an end-line character.

Characters

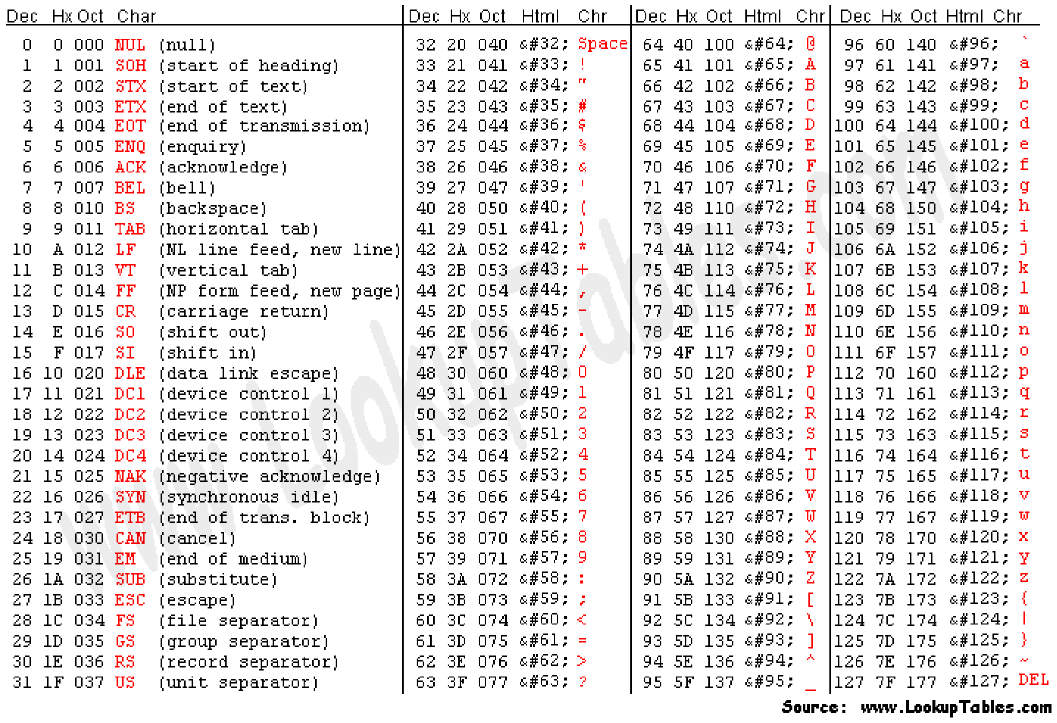
ASCII

There are many assignments of characters into numbers, but the universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world is called **ASCII** (short for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).

The ASCII assignment provides space for 256 (2^8, why? You will know later) different characters, you can find the table here (source: https://www.asciitable.com).

Some interesting values are 32 (space), 48 ('0'), 65 ('A') and 87 ('a'). The letters are arranged in the same order as in the Latin alphabet.

So, you may ask, why are there exactly 256 different characters, why can't there be 260? Remember that computer stores everything in binary form (aka 0s and 1s), 256, which is 2^8, can thus be stored by 8 bits, which is the same as 1 byte.



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

As you can see from the table, only 128 values are used, how about the 128 characters left? Is it possible for us to fit other national characters into the remaining 128 characters? The answer is no.

It is not enough for all possible languages, but it is sufficient for some.

Code points and Code pages

A code point is a number which makes a character, for example 32 is a code point which makes a *space* in ASCII encoding.

A code page is a standard for using the upper 128 code points to store specific national characters, for example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages and so on.

This means that the one and same code point can make different characters when used in different code pages. For example, the code point *200* makes 'Č', a letter used by some Slavic languages, when utilized by the ISO/IEC 8859-2 code page. Same code point *200* makes Ш (a Cyrillic letter) when used by the ISO/IEC 8849-5 code page.

In consequence, to determine the meaning of a specific code point, you have to know the target code page, in other words, the code points derived from the code page concept are ambiguous.

Unicode

Code pages helped to solve the international language problem for some time, but it soon turned out that this is not a permanent solution. Thus, **Unicode** was formed.

Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first 128 Unicode codes are identical to ASCII and the first 256 Unicode points are identical to the ISO/IEC 8849-1 code page (designed for Western European languages).

The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes.

There is more than one standard describing techniques used to implement Unicode in actual computers and computer storage systems.

UCS-4 (short for **U**niversal **C**haracter **S**et, the most general one)

It uses 32 bits (4 bytes) to store each character, and the code is just Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents.

UCS-4 is a rather wasteful standard, it increases a text's size by 4 times compared to standard ASCII.

UTF-8 (short for **U**nicode **T**ransformation **F**ormat, most common)

The concept is very smart. UTF-8 uses as many bits for each of the code points as it really needs to represent them. For example:

- All standard ASCII characters occupy 8 bits (1 byte)
- Non-Latin characters occupy 16 bites (2 bytes)
- CJK (China-Japan-Korea) ideographs occupy 24 bits (4 bytes)

Due to the features of the method used by UTF-8, BOM is not needed, but some of the tools look for it when reading the file, and many editors set it up during the save.

Python 3 fully **support** Unicode and UTF-8

You can use Unicode/UTF-8 encoded characters to name variables and other entities. You can also use them during all input and output.

Strings

Strings (brief review)

Strings in Python are **immutable** sequences.

The $len()$ function returns the length of the string

```
word = "hello"
print(len(word)) # 5
```

A string can be empty

```
word = ""
print(len(word)) # 0
```

Don't forget a backslash '\' is used as an escape character, and is not included in the string's total length

```
word = "I\'m"
print(len(word)) # 3
```

Multiline strings start and end with 3 apostrophes or 3 quotes (but the starting symbol **must** be the same as the ending). As demonstrated below:

```
multiline = '''Line 1
Line 2'''

print(len(multiline)) # 13
print(multiline)
# Line 1
# Line 2
```

There is clearly a missing character if you counted. It's a whitespace located between the two lines '$\backslash n$'. This special control character is used to force a line feed (hence its name LF), you can't see it, but it counts.

Operations on strings

Concatenation of strings

```
a = "a"
b = "b"
print(a + b) # ab
```

Replication of strings

```
a = "1"
print(a * 5) # 11111
```

As you can see, they use the same operator as numbers, the ability of doing this is called overloading.

The order in a concatenation of string matters, for the first example, exchanging $b + a$ for $a + b$ produces a different output. However, this doesn't matter in replication.

Note that one of the there should **only be 1 string** in an operation of replication of strings.

Character -> code point (also new syntax highlighting style if you've noticed)

Function: $ord()$

```
a = 'a'
b = 'α' # Greek Alpha
print(ord(a)) # 97
print(ord(b)) # 946
```

Code point -> character

Function: $chr()$

```
print(chr(97)) # a
print(chr(945)) # α
```

Note this:

```
chr(ord(x)) == x
ord(chr(x)) == x
```

More in String (brief)

Python strings are sequences, they aren't lists, but you can treat them like lists in many particular cases.

Indexing

```
a = "Hi, I am ML7!"

for i in range(len(a)):
    print(a[i], end = ' ')
# H i ,   I   a m   M L 7 !
```

Iterating

```
a = "Hi, I am ML7!"

for x in a:
    print(x, end = '_') # H_i_,_ _I_ _a_m_ _M_L_7_!_
```

Slicing

```
a = "abdefg"

print(a[1 : 3]) # bd
print(a[3 : ]) # efg
print(a[ : 3]) # abd
print(a[3 : -2]) # e
print(a[-3 : 4]) # e
print(a[ :: 2]) # adf
print(a[1 :: 2]) # beg
```

In and $not\ in$

Checks if its left argument (string) can be found anywhere within the right argument (another string)

```
ori = "abcd"

print("a" in ori) # True
print('e' in ori) # False
print("abc" in ori) # True
print("bcd" not in ori) # False
```

Python strings are **IMMUTABLE**

I've mentioned this in the brief review part, immutable simply means that you **cannot** remove any characters from the string, the only thing you can do is to remove the string as a whole.

It doesn't support $append()$ and $insert()$ either.

```
alphabet = "bcdefghijklmnopqrstuvwxy"
alphabet = "a" + alphabet
alphabet = alphabet + "z"
print(alphabet) # abcdefghijklmnopqrstuvwxyz
```

You can do this in order so combat with the problem of immutability.

Strings functions and methods

The $min()$ function returns the character with lowest code point value.

```
print(min("aAbByYzZ")) # A

s = 'Hi, my name is ML7.'
print('[' + min(s) + ']') # [ ]
```

The $index()$ function returns the index of the **first** occurrence of the string you put, note that the string **must** exist in the string, or else it will cause a $ValueError$ exception.

```
print("aAbByYzZaA".index("b")) # 2
print("aAbByYzZaA".index("Z")) # 7
print("aAbByYzZaA".index("A")) # 1
```

The $list()$ function takes its argument (string) and creates a new list containing all the string's characters, one per list element.

```
print(list("abcabc")) # ['a', 'b', 'c', 'a', 'b', 'c']
```

The $count()$ functions counts all the occurrences of the element inside the sequence, the absence of such element is fine, as it will return a value 0

```
print("abcabc".count("b")) # 2
print('abcabc'.count("d")) # 0
```

However, overlapping doesn't count, you have to write your own function / algorithm to deal with the problem

```
a = "ababa"
print(a.count("aba")) # 1
```

The $capitalize()$ function does the following, if the first character inside the string is a letter, it will be converted to upper-case, all remaining letters from the string will be converted to lower-case.

```
print("Alpha".capitalize() # Alpha
print('ALPHA'.capitalize()) # Alpha
print(' Alpha'.capitalize()) #  aplha
print('123'.capitalize()) # 123
print("αβγδ".capitalize()) # Αβγδ
```

The $center()$ function makes a copy of the original string and try to center it inside a field of a specified width, it is done by adding some spaces before and after the string.

```
print('[' + 'Test'.center(2) + ']') # [Beta]
print('[' + 'Test'.center(4) + ']') # [Beta]
print('[' + 'Test'.center(6) + ']') # [ Beta ]
```

The **two-parameter** variant of $center()$ makes use of the character from the second argument instead of a space.

```
print('[' + 'game'.center(10, '*') + ']') # [***game***]
print('[' + 'game'.center(9, '*') + ']') # [***game**]
```

Also, as shown in the above example, if the total length of the padding is odd, 1 more padding character will be added in front of the initial string.

The $endswith()$ method checks if the give string ends with the specified argument and returns True of False.

```
s = "zeta"
print(s.endswith("a")) # True
print(s.endswith("A")) # False
print(s.endswith("et")) # False
print(s.endswith("eta")) # True
```

The $startswith()$ method is a mirror reflection of $endswith()$, it checks if the given string starts with the specified substring, and returns either True of False.

```
print("omega".startswith("ega")) # False
print("omega".startswith("om")) # True
```

The $find()$ method, it is very similar to $index()$. However, there are still differences, $find()$ is safer, it doesn't generate an error for an argument containing a non-existent substring (it will return -1 then). Also, it only works with strings, don't try to apply it to any other sequence.

```
s = 'theta'
print(s.find('eta')) # 2
print(s.find('ha')) # -1
```

If you want to perform the find, not from the string's beginning, but from any position, you can use the **two-parameter** variant of the $find()$ method, as follows:

```
print('kepa'.find('a', 2)) # 3
```

The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string).

The following code points all the position in the string where the word "the" can be found

```
text = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""
```

```python
pos = text.find('the')
while pos != -1:
    print(pos)
    pos = text.find('the', pos + 1)
# prediction of output is left as an exercise
```

There is also a **three-parameter** variant of the $find()$ method, the third argument points to the first index which wont be taken in the consideration during the search (upper limit of the search)

```python
print('hello'.find('o', 1, 5)) # 4
print('hello'.find('o', 2, 4)) # -1
```

The $rfind()$ methods (**one-**, **two-** and **three-parameter**) do the same things as $find()$, but they start their searches from the end of the string, not from the beginning (hence the prefix 'r', for *right*).

```python
print("hi hi hi".rfind("hi")) # 6
print("hi hi hi".rfind("hi", 9)) # -1
print("hi hi hi".rfind("hi", 3, 9)) # 6
```

The $isalnum()$ method checks if the string contains only digits or alphabetical characters, and returns True or False.

```python
print('Python'.isalnum()) # True
print('Python3'.isalnum()) # True
print('310'.isalnum()) # True
print('@'.isalnum()) # False
print('Python_3'.isalnum()) # False
print(''.isalnum()) # False
print('ml7 michael'.isalnum()) # False
print('AβΓδ'.isalnum()) # True
print('20E1'.isalnum()) # True
```

The $isalpha()$ and $isdigit()$ methods work as their names suggest.

```python
print("Moooo".isalpha()) # True
print('Mu40'.isalpha()) # False

print('2018'.isdigit()) # True
print("Year2019".isdigit()) # False
```

The $islower()$, $isspace()$ and $isupper()$ methods also work as their names suggest.

```python
print("Moooo".islower()) # False
print('moooo'.islower()) # True

print(' \n '.isspace()) # True
print(" ".isspace()) # True
print("mooo mooo mooo".isspace()) # False
print("Moooo".isupper()) # False
print('moooo'.isupper()) # False
print('MOOOO'.isupper()) # True
```

The *join*() method, it expects on argument as a list, it must be assured that all the list's elements are strings, otherwise a *TypeError* exception will be raised. All the list's elements will be joined into one string but the string from which the method has been invoked is used as a separator. The newly created string is returned.

```python
print(" ".join(["Cristiano", "Ronaldo"])) # Cristiano Ronaldo
```

The *lower*() method makes a copy of a source string, replaces all upper-case letters with their corresponding lower-case letters, and returns the string as the result.

```python
print("SiGmA=60".lower()) # sigma=60
```

The *upper*() method exists as well.

```python
print("SiGmA=60".upper()) # SIGMA=60
```

The *swapcase*() method makes a new string by swapping the case of all letters within the source string, lower-case to upper-case and vice versa.

The *lstrip*() method returns a newly created string formed from the original one by removing all **leading** whitespaces.

```python
print("[" + "  ml7 ".lstrip() + "]") # [ml7 ]
```

The *strip*() method combines the effects caused by *lstrip*() and *rstrip*(), it makes a new string lacking all the leading and trailing whitespaces.

```python
print("[" + "   alpha   ".strip() + "]") # [alpha]
```

The **one-parameter** *lstrip*() method does the same as its parameter-less version, but removes all characters enlisted in its argument (a string), not just whitespaces.

```python
print("www.youtube.com".lstrip("w.")) # youtube.com
```
Mind the word **leading**.

```python
print("www.youtube.com".lstrip(".com")) # www.youtube.com
```

The rstrip() method exist, but affect the opposite side of the string.

```python
print("[" + "  ml7  ".rstrip() + "]") # [  ml7]
print("www.youtube.com".rstrip(".com")) # www.youtube
```

The *replace*() method (**two-parameter**) returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument. (AND YES YOU CAN REMOVE CHARACTERS FROM THE STRING!!!)

```python
print("facebook.com".replace("facebook", "instagram")) # instagram.com
print("Hello World".replace(" World", "")) # Hello
```

The *replace*() method (**three-parameter**) limits the number of replacements.

```python
print("hello, hello".replace("hello", "Hola", 1)) # Hola, hello
print("hello, hello".replace("hello", "Hola", 2)) # Hola, Hola
```

The $split()$ method does what is says, it splits the string and builds a list of all detected substrings. It assumes that the substrings are delimited by whitespaces, the spaces don't take part in the operation, and aren't copied into the resulting list. Empty string will result in an empty list. The reverse operation can be performed by the $join()$ method.

```
print("ml7 michael".split()) # ['ml7', 'michael']
print("ml7     michael".split()) # ['ml7', 'michael']
```

The $swapcase()$ method

```
print("I know that I know nothing.".swapcase())
# i KNOW THAT i KNOW NOTHING.
```

The $title()$ method changes ever word's first letter to upper-letter, turning all other ones to lower-case.

```
print("I know that I know nothing. Part 1.".title())
# I Know That I Know Nothing. Part 1.
```

String Comparisons

Some comparing operators with integers work in strings as well, that includes:

```
==
!=
>
>=
<
<=
```

Don't forget that Python is not aware of subtle linguistic issues, it just compares **code point** values, character by character.

For examples

```
("alpha" == "alpha") # True
("alpha" != "Alpha") # True
("alpha" < "Alpha") # False
```

What if the strings have different length?

If there are two strings, denote as a and b, and let's assume a is a prefix of b (b starts with a), then b is larger **or** equal to a. If the length of string b is equal to that of string a, then the two strings are identical. Or else, string b is larger than string a. Same theory applies to $smaller$, $smaller\ or\ \ equal\ to$, $larger$, $larger\ or\ equal\ to$ operators.

Even if a string contains digits only, it's still **not a number**, for example, this holds:

$$"10" != "010"$$

Comparing strings against numbers are not recommended (and why would you do that?)

They only comparisons you can perform with impunity are $==$ and $!=$ operators. Using anyone of the other comparison operators will raise a $TypeError$ exception.

String Sorting

Comparing is closely related to sorting, there are two possible ways to sort **lists** containing string. Sorting is very common in the real world, anytime you see a list of names, goods, titles, cities, countries, you expect them to be sorted.

Method 1

The function $sorted()$ will be used, as you can see in the example below, the original list is not changed, a new list was returned. (This description is a bit simplified compared to the actual implementation, we'll discuss it later)

```
a = ["delta", "charlie", "echo", "bravo", "alfa"]
b = sorted(a)
print(a) # ['delta', 'charlie', 'echo', 'bravo', 'alfa']
print(b) # ['alfa', 'bravo', 'charlie', 'delta', 'echo']
```

Method 2

The ordering is now preformed in situ by the method named $sort()$.

```
a = ["delta", "charlie", "echo", "bravo", "alfa"]
a.sort()
print(a) # ['alfa', 'bravo', 'charlie', 'delta', 'echo']
```

Conversion between Strings and Numbers

The conversion from numbers to strings are always possible, all you have to do is this:

```
pi = 3.14
s = str(pi)
print(s) # 3.14
print(type(s)) # <class 'str'>
```

The conversion from strings to numbers is possible when and only when the string represents a valid number, if it's not, a $ValueError$ exception will be shown.

```
s1 = "5"
a = int(s1)
print(a) # 5
s2 = "3.141592"
pi = float(s2)
print(s2) # 3.141592
s3 = "abc"
b = int(s3) # ValueError
```

Some takeaways

1. Strings are key tools in modern data processing, as most useful data are actually strings. For example, using a web search engine (which seems quite trivial these days) utilizes extremely complex and complicated string processing, involving unimaginable amounts of dat.
2. Comparing strings in a strict way (as Python does) can be very unsatisfactory when it comes to advanced searches (during extensive database queries). Responding to this demand, a number of *fuzzy* string comparison algorithms has been created and implemented. These algorithms are able to find strings which aren't equal in the Python sense, but similar, two examples are **Hamming distance** (Hamming distance - Wikipedia) and **Levenshtein distance** (Levenshtein distance - Wikipedia).

3. Another way of comparing strings is finding their *acoustic* similarity (like “raise” and “race”). Such similarity has to be established for every language (or even dialect). An algorithm used to perform such a comparison for English is called **Soundex** (Soundex - Wikipedia) and was invented in 1918!
4. Due to limited native float and integer data precision, it’s sometimes reasonable to store and process huge numeric values as strings. This is the technique Python uses when you force it to operate on an integer number consisting of a very large number of digits. ***In Python, value of an integer is not restricted by the number of bits, but by the available memory.*** (Source: Built-in Types — Python 3.10.4 documentation, a more detailed explanation:)

Anything that can go wrong, will go wrong. Murphy’s law

Here is a simple program calculating the square root of a number

```
import math
x = float(input())
y = math.sqrt(x)
print(y)
```

However, if you didn’t input a valid number, you will get a $ValueError$. As to how to fix this problem, please refer to $Exceptions\ in\ Python$.