

Podstawy programowania

rozdział 6: **LICZBY RZECZYWISTE** **I** **NAPISY**

ostatnia modyfikacja: 04.12.18

Podstawy programowania

Opowiemy o tym
**jak komputery przechowują liczby
rzeczywiste**

i będzie to też opowieść o
osobliwościach komputerowej arytmetyki

Podstawy programowania

liczby zmiennoprzecinkowe:

- implementację arytmetyki zmiennoprzecinkowej standaryzuje norma **IEEE-754** pochodząca z 1985 roku
- definiuje ona dwa formaty danych:
 - single → 32 bity (**pojedyncza** precyzja)
 - double → 64 bity (**podwójna** precyzja)

Podstawy programowania

liczby zmiennoprzecinkowe:

- wartość przechowywana jest **inaczej**, niż w danych całkowitych
- nie jako **pojedyncza** wartość zakodowana dwójkowo...
- a jako **trójka** danych
 - znak (+ / -)
 - cecha
 - mantysa

Podstawy programowania

o co chodzi z tą cechą i mantysą?

- średnia odległość od Ziemi do Słońca:
149 600 000 000 m → musimy użyć 12 cyfr
- zapiszmy to inaczej: $1496 \cdot 10^8$ m → tylko 5 cyfr!

Podstawy programowania

1496 · 10⁸

mantysa

nośnik dokładności

cecha

nośnik skali

Podstawy programowania

liczby zmiennoprzecinkowe pojedynczej precyzji:

- struktura:
 - znak liczby: 1 bit
 - cecha: 8 bitów
 - mantysa: 23 bity
- dokładność – ok. 7÷8 cyfr znaczących
- zakres reprezentacji:
od $1,18 \cdot 10^{-38}$ do około $3.4 \cdot 10^{38}$

Podstawy programowania

liczby zmiennoprzecinkowe podwójnej precyzji:

- struktura:
 - znak liczby: 1 bit
 - cecha: 11 bitów
 - mantysa: 52 bity
- dokładność – ok. 15÷16 cyfr znaczących
- zakres reprezentacji:
od $2,23 \cdot 10^{-308}$ do około $1.80 \cdot 10^{308}$

Podstawy programowania

o co chodzi z tą liczbą cyfr znaczących?

- wykonajmy pewne trywialne dodawanie dwóch liczb rzeczywistych – ile to będzie?

```
>>>print(1234567890.0 + 0.987654321)
```

Podstawy programowania

```
>>>print(1234567890.0 + 0.987654321)  
1234567890.9876542
```



SIC!

Podstawy programowania

- eksperyment pozornie bezcelowy:

```
liczby=[70000000000.,7000000000.,700000000.,7000000.,700000.,  
        70000.,7000.,700.,70.,7.,.7,.07,.007,.0007,.00007,  
        .000007,.0000007,.00000007,.000000007,.0000000007]  
  
suma1 = suma2 = 0.0  
for i in range(len(liczby)):  
    suma1 += liczby[i]  
for i in reversed(range(len(liczby))):  
    suma2 += liczby[i]  
print(suma1)  
print(suma2)
```

Podstawy programowania

szok!

```
77707777777.77779  
77707777777.77777
```


Podstawy programowania

oznacza to, że komputerowa arytmetyka liczb rzeczywistych **nie jest**:

- **przemienna:**

$$a + b + c \neq c + a + b$$

- **łączna:**

$$(x + y) + z \neq x + (y + z)$$

$$(x \cdot y) \cdot z \neq x \cdot (y \cdot z)$$

- **rozdzielna**

$$x \cdot (y + z) \neq x \cdot y + x \cdot z$$

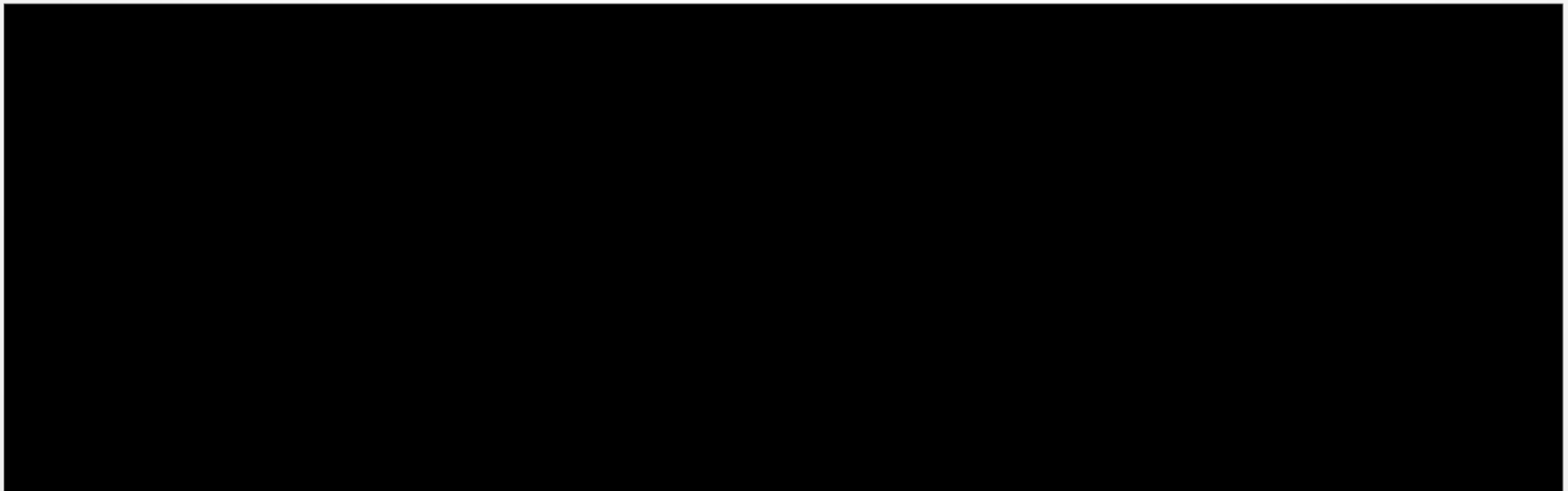
Podstawy programowania

przerażający eksperyment – co tu się zaraz stanie?

```
x = 1
while x != x + 1:
    x *= 2
print(x)
```

Podstawy programowania

nic się nie stanie – program się zapętlili :(



Podstawy programowania

jeszcze raz to samo, ale na liczbach rzeczywistych:

```
x = 1.0
while x != x + 1.0:
    x *= 2.0
print(x)
```

Podstawy programowania

szok i niedowierzenie!

9007199254740992.0

Podstawy programowania

sprawdźmy:

```
>>> print(9007199254740992.0 + 1.0)  
9007199254740992.0
```



SIC!

Podstawy programowania

zapamiętaj:

- liczby rzeczywiste są **świetnym i użytecznym** wynalazkiem
- ale trzeba znać ich **słabe punkty** i wiedzieć, jak je omijać
- w każdym razie: nie używaj liczb rzeczywistych do przechowywania **danych finansowych**

Podstawy programowania

- **rakieta Ariane 5**
- uległa katastrofie 4 czerwca 1996 w 37 sekundzie od startu
- przyczyną był błąd w oprogramowaniu rakiety, dotyczący obliczeń na liczbach rzeczywistych
- uznaje się, że był to najdroższy „bug” w historii – kosztował ok. **370 mln dolarów**



Podstawy programowania

Napisy

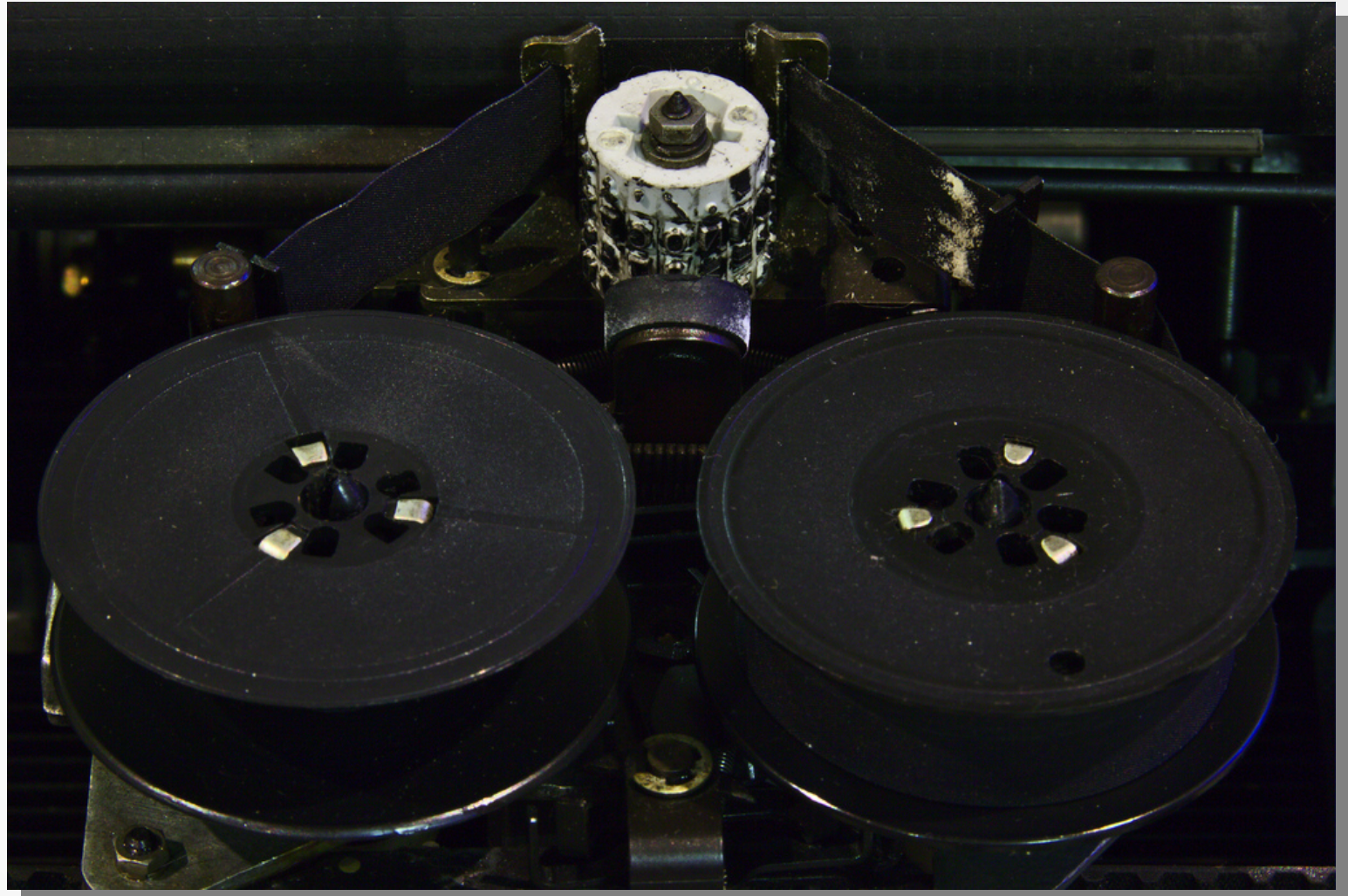
Podstawy programowania

- **Pytanie:
co to jest?**



Podstawy programowania

- carriage
czyli
karetka



Podstawy programowania

pytanie: jak komputery przechowują znaki?

odpowiedź: jako liczby

Podstawy programowania

przedstawiamy: kod ASCII

American Standard Code for Information Interchange

- uznawana powszechnie norma międzynarodowa, opisująca jakie wartości numeryczne przypisuje się konkretnym znakom
- reprezentuje jeden znak na jednym bajcie
- zastąpił liczne inne kody, stosowane przez różnych producentów na całym świecie (np. kod o nazwie EBCDIC, stosowany w komputerach firmy IBM)

Podstawy programowania

kod ASCII

- pierwotnie ASCII był kodem siedmiobitowym (spadek po dalekopisach) → 128 różnych znaków
- obecnie ASCII jest kodem ośmiobitowym → 256 różnych znaków
- stworzony dla alfabetu łacińskiego, brak w nim znaków narodowych (np. polskich)
- jest w nim za to spory zestaw znaków specjalnych:
!@#\$%^&*()_+ - = : " ' { } [] \ | < > ? , . /

Podstawy programowania

kod UNICODE (unikod)

- w zastosowaniach międzynarodowych kod ASCII zastępowany jest przez tzw. **UNICODE**
- obecnie UNICODE definiuje kodowanie dla ponad 120 tys. znaków z 129 języków (żywych i martwych)
- najczęściej kodowany jako tzw. **UTF-8**: znaki ASCII koduje się na 8 bitach, znaki spoza ASCII na 16 bitach
- ASCII jest **podzbiorem** UNICODE

Podstawy programowania

kod ASCII

- stosowany wszędzie tam, gdzie używa się informacji znakowej – w komputerach wszelkiej maści, wiadomościach SMS, kartach SIM, kartach bankomatowych, czytnikach kodów paskowych, wyświetlacza alfanumerycznych, urządzeniach gospodarstwa domowego, ...

Podstawy programowania

kod ASCII – ważne zastrzeżenie!

- dla komputera zawartość dowolnego bajtu w pamięci jest **amorficzna** (bezpostaciowa) – nie nadaje się jej żadnej szczególnej interpretacji
- w szczególności, w tzw. **architekturze von Neumanna** każdy bajt może zawierać daną albo kod
- jeśli to jest dana, to może być liczbą albo znakiem – interpretacja zależy od **programu**, nie od **komputera**

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT End of transmission	0x24	36	\$	0x44	68	D	0x64	100	d
0x05	5	ENQ Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1 Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2 Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3 Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4 Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS File separator	0x3C	60	<	0x5C	92	\	0x7C	124	
0x1D	29	GS Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Podstawy programowania

kod ASCII:

- pierwsze 32 znaki to tzw. **kody sterujące**
- **spacja** ma kod 32
- **cyfry** kodowane są spójnie kodami od 48 (0) do 57 (9)
- **wielkie litery** kodowane są spójnie: od 65 (A) do 90 (Z)
- **małe litery** kodowane są spójnie: od 97 (a) do 122 (z)
- ciekawostka: kody litery małej i wielkiej różnią się o spację:
np. $88 (\text{X}) + 32 = 120 (\text{x})$

Podstawy programowania

napisy w Pythonie

- w terminologii angielskiej: „string” (dosł. sznur, struna)
- w terminologii polskiej: „łańcuch”, „napis”

Podstawy programowania

napisy w Pythonie

- aby pewien ciąg znaków został uznany za napis, należy go ująć w **apostrofy** albo **cudzysłowy**
- dotyczy to również pojedynczych znaków
- napis może zostać podstawiony do zmiennej tak, jak każda inna dana

Podstawy programowania

zmienna o nazwie „imię” przechowuje ciąg znaków składających się na słowo
Nabuchodonozor

imię = 'Nabuchodonozor'
nazwisko = "Kowalski"

zmienna o nazwie „nazwisko” przechowuje ciąg znaków składających się na słowo
Kowalski

Podstawy programowania

napisy w Pythonie

- napisy mogą być również elementami list
- co więcej, niektóre elementy list mogą być napisami, a inne liczbami

Podstawy programowania

```
ekipa = ['Lolek', 'Bolek', 'Tola']
```

```
gang = ["Alibaba", "Rozbójnik"]
```

```
mix = [ 1, 'tekst' ]
```


Podstawy programowania

- **print()** bez zastrzeżeń akceptuje zmienne napisowe

```
x = 'ała ma kota'  
print(x)  
y = "Natenczas Wojski chwycił"  
print(y)
```

```
ała ma kota  
Natenczas Wojski chwycił
```

Podstawy programowania

- **funkcja len() ukazuje swoją drugą twarz:**

```
x = 'ala ma kota'  
print(len(x))
```

Podstawy programowania

- **napis może być pusty (jak lista)**

```
x = ''  
print(len(x))
```

0

Podstawy programowania

jeśli potrzebujesz apostrofu w napisie ograniczonym cudzysłowami, to nie ma problemu
(uwaga – slajd zawiera lokowanie produktu)

```
y = "Idę do McDonald's"  
print(y)
```

Idę do McDonald's

Podstawy programowania

jeśli potrzebujesz cudzysłowu w napisie ograniczonym apostrofami, to też nie ma problemu

```
z = 'Lubię "Latający cyrk Monty Pythona"'
print(z)
```

```
Lubię "Latający cyrk Monty Pythona"
```

Podstawy programowania

znak \ (backslash)

- używa się go wewnątrz napisów do „cytowania”, tzn. poprzedzania pewnych znaków w celu potraktowania ich w **szczególny** sposób
- dzięki temu można **wplatać** cudzysłowy i apostrofy tam, gdzie normalnie nie byłoby to możliwe

Podstawy programowania

cytowanie apostrofu:

```
y = 'Idę do McDonald\'s'  
print(y)
```

Idę do McDonald's

Podstawy programowania

bez cytowania będzie problem...

```
y = 'Idę do McDonald's'
```

```
SyntaxError: invalid syntax
```


Podstawy programowania

zapamiętaj:

użycie `\` dla cytowania daje w wyniku jeden znak!

```
y = '\\'
print(len(y))
```

1

Podstawy programowania

cytowanie cudzysłowu:

```
z = "Lubię \"Latający cyrk Monty Pythona\""  
print(z)
```

Lubię "Latający cyrk Monty Pythona"

Podstawy programowania

napis wielowierszowy:

```
z = """Co to za zwierzę,  
powiedz mi proszę,  
róg ma na nosie,  
nogi ma w rosie."""  
print(z)
```

```
Co to za zwierzę,  
powiedz mi proszę,  
róg ma na nosie,  
nogi ma w rosie.
```

Podstawy programowania

napis wielowierszowy – tak też można:

```
z = '''Co to za zwierzę,  
trudno mi orzec:  
czy nosorożec,  
czy rosonożec.'''  
print(z)
```

```
Co to za zwierzę,  
trudno mi orzec:  
czy nosorożec,  
czy rosonożec.
```

Podstawy programowania

znak \ (backslash)

- może być również użyty do zapisu tych kodów ASCII, które nie mają czytelnej reprezentacji znakowej
- są to między innymi tzw. „białe znaki” (ang. *white spaces*), ale nie tylko

Podstawy programowania

znak \t

kod ASCII: 9 (0x09)

nazwa: HT → Horizontal Tabulation

- powodował, że karetką drukującą przeskakiwała do ustalonych kolumn, tzw. „koników tabulatora”
- w ten sposób wyjaśniła się tajemnica działania klawisza Tab w edytorze

Podstawy programowania

działanie tabulatora:

```
napis = "Ala\tma\tkota"  
print('123456789012345678901234567890')  
print(napis)
```

```
123456789012345678901234567890  
Ala      ma      kota
```

Podstawy programowania

znak \b

kod ASCII: 8 (0x08)

nazwa: BS → BackSpace

- powodował, że karetką drukującą cofała się o jeden znak

Podstawy programowania

działanie znaku backspace:
(może nie działać w konsoli Pythona :()

```
napis = "Ala\bma\bkota"  
print('123456789012345678901234567890')  
print(napis)
```

```
123456789012345678901234567890  
Almkota
```

Podstawy programowania

- uwaga: to nie jest tak, że `\b` usuwa jakiegolwiek znaki z napisu – on tylko **cofa** karetkę!

```
napis = "Ala\bma\bkota"  
print(len(napis))
```

Podstawy programowania

znak \r

kod ASCII: 13 (0x0D)

nazwa: CR → Carriage Return

- powodował, że karetką wracała do początku linii

Podstawy programowania

- działanie powrotu karetki

```
napis = "Ala\rma"  
print(napis)
```

```
maa
```

Podstawy programowania

znak \n

kod ASCII: 10 (0x0A)

nazwa: NL → New Linie

- powodował, że dalekopis wysuwał papier o jeden rząd znaków, tym samym przenosząc karetkę do następnego wiersza

Podstawy programowania

- działanie nowej linii

```
napis = "Ala\nma"  
print(napis)
```

```
Ala  
ma
```

Podstawy programowania

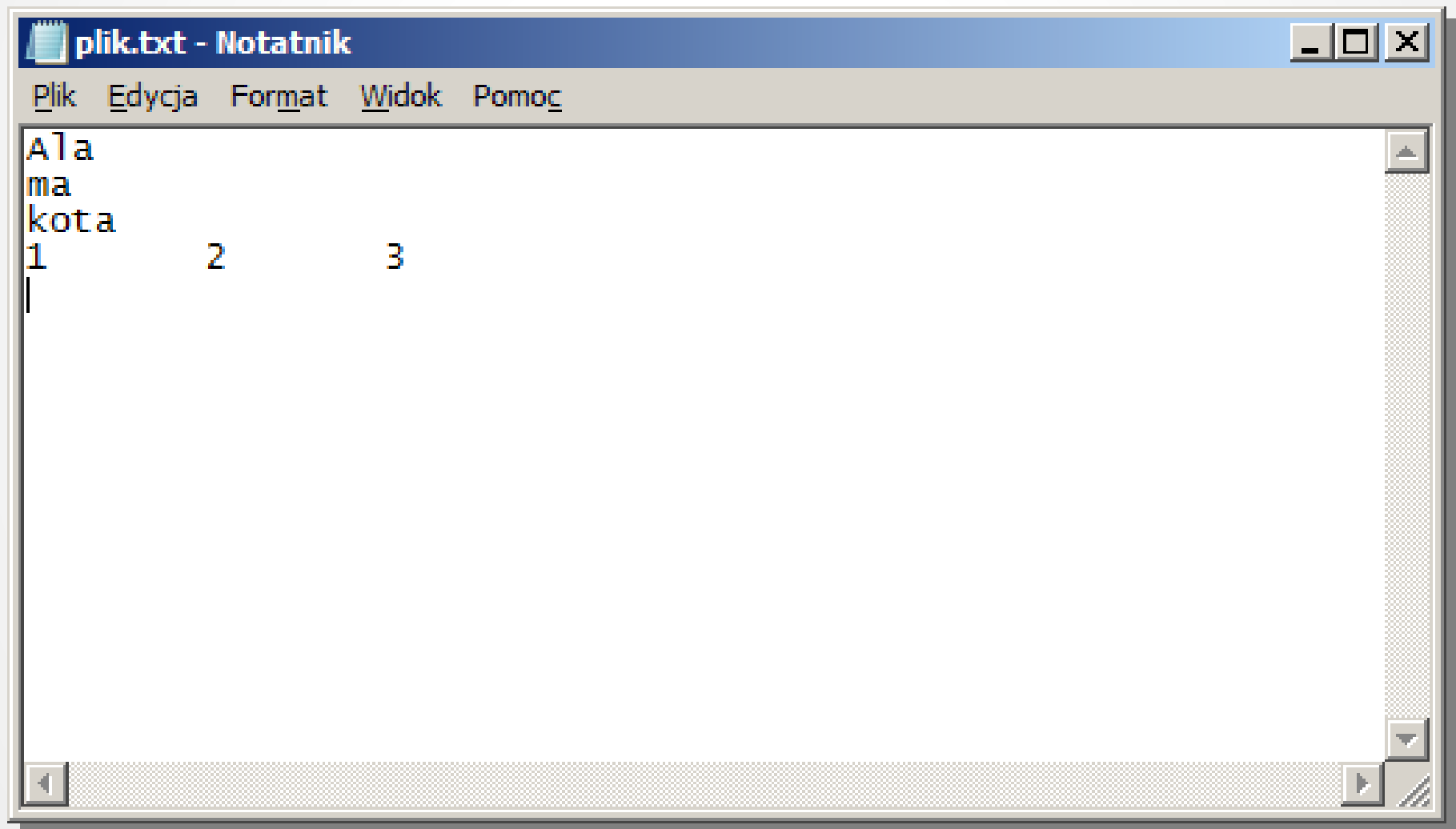
zapamiętaj:

znaki CR i LF używane są w plikach tekstowych do oznaczania końców wierszy

- pokażmy to na prostym przykładzie

Podstawy programowania

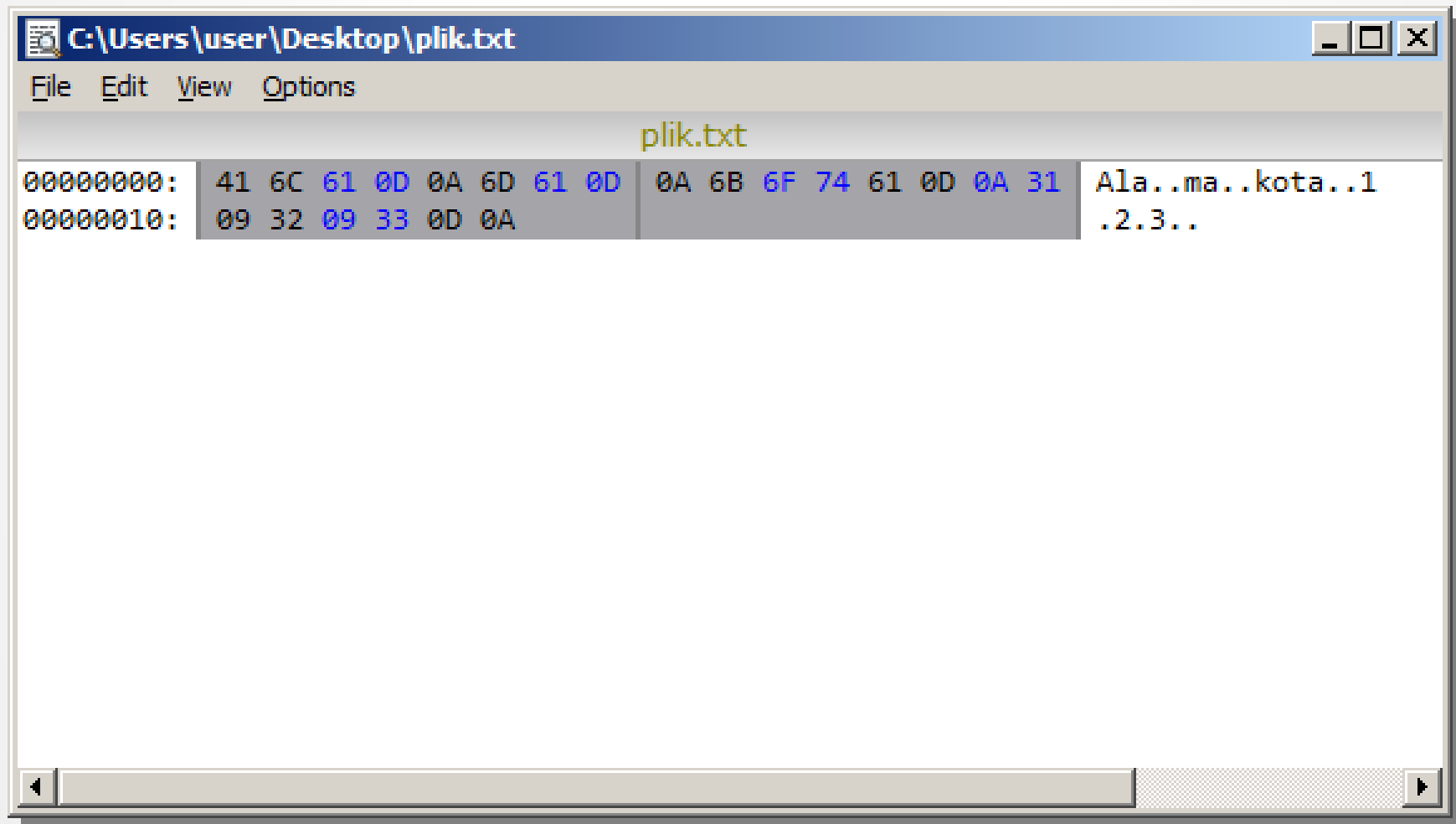
Oto plik tekstowy w postaci, w jakiej oglądamy go np. w Notatniku:



```
Ala
ma
kota
1
|
```


Podstawy programowania

A tak wygląda ten plik, kiedy zajrzemy do jego wnętrza:



The screenshot shows a Windows Notepad window titled "C:\Users\user\Desktop\plik.txt". The menu bar includes "File", "Edit", "View", and "Options". The text area displays the contents of the file in hexadecimal and ASCII format. The first line of data is "00000000: 41 6C 61 0D 0A 6D 61 0D 0A 6B 6F 74 61 0D 0A 31", which corresponds to the ASCII string "Ala..ma..kota..1". The second line of data is "00000010: 09 32 09 33 0D 0A", which corresponds to the ASCII string ".2.3..".

```
plik.txt
00000000: 41 6C 61 0D 0A 6D 61 0D 0A 6B 6F 74 61 0D 0A 31 Ala..ma..kota..1
00000010: 09 32 09 33 0D 0A .2.3..
```

Podstawy programowania

zapamiętaj:

- z powodu bardzo specyficznej roli, jaką pełni w napisach znak ****, aby umieścić go w napisie samodzielnie, należy go **zacytować** (czyli podwoić)

Podstawy programowania

- cytowanie cytowania

```
napis = "Ten znak \\ to backslash"  
print(napis)
```

Ten znak \ to backslash

Podstawy programowania

- **operatory napisowe**
- do manipulowania napisami używamy (tak jak w przypadku liczb) **operatorów**
- jak się za chwilę okaże, może być i tak, że jeden i ten sam operator wykonuje **inną** czynność na liczbach, a inną na napisach
- taką sytuację nazywamy **przeciążeniem operatora** (ang. *operator overloading*)

Podstawy programowania

operator konkatencji: + (plus)

- łączy lewy i prawy argument w jeden nowy napis
- np:
 'py' + 'thon' → 'python'
 'thon' + 'py' → 'thonpy'

Podstawy programowania

operator powielania: * (gwiazdka)

- powiela jeden argument tyle razy, ile określono w drugim argumencie, dając w efekcie nowy napis

- np:

'py' * 2 → 'pypy'
10 * '*' → '*****'

Podstawy programowania

funkcja ord(x):

jak ordinal → porządkowy

- zamienia znak na odpowiadający mu kod
- ord(x):
 - argument: pewien znak
 - wynik: jego kod
 - efekt: żaden
- np:
`print(ord('A')) → 65`

Podstawy programowania

funkcja chr(x):

jak character → znak

- zamienia kod na odpowiadający mu znak
- chr(x):
 - argument: kod pewnego znaku
 - wynik: ten znak
 - efekt: żaden
- np:
`print(chr(65))` → A

Podstawy programowania

- **Sequence types czyli sekwencje**
- typy danych służące do kolekcjonowania danych w **uporządkowany** sposób noszą w Pythonie angielską nazwę „sequence types” czyli **sekwencje**
- sekwencjami są **listy**
- sekwencjami są także **łańcuchy (napisy)**
- wszystkie sekwencje zachowują się w podobny sposób, mają wiele **wspólnych** funkcji i metod

Podstawy programowania

- **Dane „immutable” (niezmienne)**
- niektóre dane dostępne w Pythonie są traktowane jako „immutable” (niezmienialne)
- oznacza to, że **nie wolno zmieniać ich zawartości**
- łańcuchy w Pythonie są *immutable*
- można jednak dowolnie tworzyć **nowe wartości** bazując na **wartościach już dostępnych**

Podstawy programowania

Dane „immutable” (niezmienne)

- fakt, że łańcuchy w Pythonie są *niezmienne* pociąga za sobą kilka konsekwencji:

Podstawy programowania

Po pierwsze:

- instrukcja `del` nie działa na znaki w napisach – jej użycie jest w tym kontekście niedozwolone!
- można jednak wykonać ją w odniesieniu do całego napisu

Podstawy programowania

- **Po drugie:**

łańcuchy nie mają metody `append()`

Podstawy programowania

- **Po trzecie:**

łańcuchy nie mają metody `insert()`

Podstawy programowania

- Ale wszystko, co wiemy o wycinkach, ma również zastosowanie dla łańcuchów:

```
napis = "abdefg"  
print(napis[1])  
print(napis[-1])  
print(napis[1:3])  
print(napis[3:])  
print(napis[:3])
```

```
b  
g  
bd  
efg  
abd
```

Podstawy programowania

- wykorzystamy tę okazję, aby pokazać wycinek z krokiem:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print(napis[0:10:1])  
print(napis[::1])  
print(napis[1:18:2])  
print(napis[:-1:3])  
print(napis[1:-1:4])
```

```
abcdefghij  
abcdefghijklmnopqrstuvwxyz  
bdfhjlnpr  
adgjmpsvy  
bfjnrv
```


Podstawy programowania

- Operator **in** działa tak, jak tego oczekujemy:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('f' in napis)  
print('F' in napis)  
print('1' in napis)
```

```
True  
False  
False
```

Podstawy programowania

- Operator **not in** również:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('f' not in napis)  
print('F' not in napis)  
print('1' not in napis)
```

```
False  
True  
True
```

Podstawy programowania

- Lewym argumentem operatorów **in** i **not in** mogą być również napisy:

```
napis = "abcdefghijklmnopqrstuvwxyz"  
print('ghi' not in napis)  
print('ghi' in napis)  
print('Xyz' not in napis)  
print('xyz' in napis)
```

```
False  
True  
True  
True
```

Podstawy programowania

funkcja min(sekw):

jak minimum

- podaje najmniejszy element sekwencji
- min(sekw):
 - argument: pewna sekwencja
 - wynik: najmniejszy element w sekwencji
- np:
`print(min('aAbByYzZ')) → A`

Podstawy programowania

- kilka przykładów działania funkcji `min()`:

```
t = 'Ala ma kota'
print('[' + min(t) + ']')
t = "aqżŻ"
print(min(t))
t = [0,1,2]
print(min(t))
```

```
[ ]
a
0
```

Podstawy programowania

funkcja max(sekw):

jak maximum

- podaje największy element sekwencji
- max(sekw):
 - argument: pewna sekwencja
 - wynik: największy element w sekwencji
- np:
`print(max('aAbByYzZ')) → z`

Podstawy programowania

- kilka przykładów działania funkcji `max()`:

```
t = 'Ala ma kota'  
print(max(t))  
t = "aqżŻ"  
print(max(t))  
t = [0,1,2]  
print(max(t))
```

```
t  
ż  
2
```

Podstawy programowania

- **Uwaga:**

w kodzie ASCII wielkie litery **leżą przed małymi**,
więc wielkie **A** jest „mniejsze” od małego **a**

Podstawy programowania

metoda `index(elem)`:

- podaje położenie elementu w sekwencji
- `sekw.index(elem)`:
 - argument: pewien element
 - wynik: położenie elementu w sekwencji
- np:
`print('aAbByYzZ'.index('b')) → 2`

Podstawy programowania

- przykłady działania metody `index()`:

```
t = 'Ala ma kota'  
print(t.index('t'))  
print(t.index('A'))  
print(t.index('a'))
```

```
9  
0  
2
```

Podstawy programowania

- **Uwaga:**

próba znalezienia położenia elementu, którego w sekwencji nie ma, skończy się **katastrofą**:

```
t = 'Ala ma kota'  
i = t.index('z')
```

```
Traceback (most recent call last):  
  File "x.py", line 2, in <module>  
    i = t.index('z')  
ValueError: substring not found
```

Podstawy programowania

- dalsze przykłady działania metody `index()`:

```
t = [0,0,1,1,2,2]
print(t.index(0))
print(t.index(1))
print(t.index(2))
```

```
0
2
4
```

Podstawy programowania

metoda `count(elem)`:

jak count → liczyć

- zlicza wystąpienie elementu w sekwencji
- `sekw.count(elem)`:
 - argument: pewien element
 - wynik: liczba wystąpień elementu w sekwencji
- np:
`print('abccabc'.count('b'))` → 2

Podstawy programowania

- przykłady działania metody `count()`:

```
t = 'Ala ma kota'  
print(t.count('t'))  
print(t.count('a'))  
print(t.count(' '))
```

```
1  
3  
2
```

Podstawy programowania

- **Uwaga:**

próba policzenia wystąpień elementu, którego w sekwencji nie ma, kończy się **pomyślnie**:

```
t = 'Ala ma kota'  
i = t.count('z')  
print(i)
```

0

Podstawy programowania

Metody i funkcje łańcuchowe (stricte!)

Podstawy programowania

metoda `capitalize()`:

jak `capitalize` → „kapitalikuj”

- zwraca kopię łańcucha z pierwszym znakiem zamienionym na wielką literę (o ile to możliwe), a kolejnymi zamienionymi na małe litery (o ile to możliwe)
- `str.capitalize()`:
 - argument: żaden
 - wynik: kopia `str`, ale z wielką literą na początku i małymi w dalszym ciągu
- np:
`print('ala'.capitalize())` → `Ala`

Podstawy programowania

- przykłady działania metody `capitalize()`:

```
print('Ala'.capitalize())  
print('ALA'.capitalize())  
print(' Ala'.capitalize())  
print('123'.capitalize())  
print("Łódź".capitalize())
```

```
Ala  
Ala  
 Ala  
123  
Łódź
```

Podstawy programowania

metoda `center(szer)`:

jak center → „wycentruj”

- zwraca kopię łańcucha wycentrowanego w polu o szerokości `szer`
- `str.center(szer)`:
 - argument: szerokość docelowego pola
 - wynik: kopia `str` dopasowana do pola o podanej szerokości
- np:
`print('['+'ala'.center(5)+''])` → `[ ala ]`

Podstawy programowania

- przykłady działania metody `center()`:

```
print('[' + 'Ala'.center(2) + ']')  
print('[' + 'Ala'.center(3) + ']')  
print('[' + 'Ala'.center(4) + ']')
```

```
[Ala]  
[Ala]  
[Ala ]
```

Podstawy programowania

metoda `center(szer,wypełn):`

- działa jak `center`, ale zamiast spacji używa wypełniacza `wypełn`
- `str.center(szer,wypełn):`
 - argumenty: `szer`: szerokość docelowego pola
`wypełn`: wypełniacz (dokładnie jeden znak!)
 - wynik: kopia `str` dopasowana do pola o podanej szerokości z użyciem wypełniacza `wypełn`
- np:
`print('ala'.center(5, '*'))` → `*ala*`

Podstawy programowania

metoda `endswith(przyrostek)`:

jak „ends with” → „kończy się na”

- sprawdza, czy łańcuch kończy się ciągiem znaków przyrostek
- `str.endswith(przyrostek)`:
 - argumenty: przyrostek: łańcuch
 - wynik: True / False
- np:
`print('Ala ma kota'.endswith('ta'))` → True

Podstawy programowania

- przykłady działania metody `endswith()`:

```
t = 'Przychodzi pyton do lekarza'  
print(t.endswith('a'))  
print(t.endswith('rza'))  
print(t.endswith('rz'))
```

```
True  
True  
False
```

Podstawy programowania

metoda `find(podłańcuch)`:

jak „find” → „znajdź”

- szuka podłańcucha w łańcuchu (podobnie jak `index()`, ale bezpieczniej)
- `str.find(podłańcuch)`:
 - argumenty: podłańcuch: szukany łańcuch
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('ma')) → 4`

Podstawy programowania

- przykłady działania metody `find()`:

```
t = 'Przychodzi pyton do lekarza'  
print(t.find('ton'))  
print(t.find('Przy'))  
print(t.find('prz'))
```

```
13  
0  
-1
```

Podstawy programowania

- **Uwaga:**

nie używaj `find()`, jeśli tylko chcesz sprawdzić, czy pewien znak **występuje w łańcuchu** – operator `in` będzie szybszy!

Podstawy programowania

metoda `find(podłańcuch, start)`:

jak „find” → „znajdź”

- szuka podłańcucha w łańcuchu od wskazanej pozycji
- `str.find(podłańcuch, start)`:
 - argumenty: podłańcuch: łańcuch
start: pozycja, od której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('a', 4)) → 5`

Podstawy programowania

metoda `find(podłańcuch, start, meta)`:

jak „find” → „znajdź”

- szuka podłańcucha w łańcuchu od pierwszej wskazanej pozycji, ale nie dalej niż druga wskazana pozycja (jak w wycinkach)
- `str.find(podłańcuch, start, meta)`:
 - argumenty: podłańcuch: łańcuch
start: pozycja, od której należy szukać
koniec: pozycja, do której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('Ala ma kota'.find('a', 4, 5)) → -1`

Podstawy programowania

metoda `isalnum()`:

jak „is alphanumeric” → „czy alfanumeryczny”

- sprawdza, czy łańcuch zawiera wyłącznie litery i cyfry
- `str.isalnum()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Ala ma kota'.isalnum())` → False

Podstawy programowania

- przykłady działania metody `isalnum()`:

```
t = 'Sześć łabędzi'  
print(t.isalnum())  
t = 'łabędź'  
print(t.isalnum())  
t = ''  
print(t.isalnum())
```

```
False  
True  
False
```

Podstawy programowania

metoda `isalpha()`:

jak „is alphabetic” → „czy alfabetyczny”

- sprawdza, czy łańcuch zawiera wyłącznie litery
- `str.isalpha()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Alamakota'.isalpha()) → True`

Podstawy programowania

metoda `isdigit()`:

jak „is digit” → „czy cyfra”

- sprawdza, czy łańcuch zawiera wyłącznie cyfry dziesiętne
- `str.isdigit()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('20160101'.isdigit()) → True`

Podstawy programowania

metoda `islower()`:

jak „is lower” → „czy mała litera”

- sprawdza, czy łańcuch zawiera wyłącznie małe litery
- `str.lower()`:
 - argument: żaden
 - wynik: True / False
- np:
`print('Alamakota'.islower())` → False

Podstawy programowania

metoda `isspace()`:

jak „is space” → „czy spacja”

- sprawdza, czy łańcuch zawiera wyłącznie tzw. białe znaki
- `str.isspace()`:
 - argument: żaden
 - wynik: True / False
- np:
`print(' \t\r\n'.isspace()) → True`

Podstawy programowania

metoda `join(lista)`:

jak „join” → „złącz”

łączy elementy listy w jeden napis, rozdzielając je łańcuchem `str`;
elementy listy **muszą być napisami!**

- `str.join(list)`:
 - argument: `list`: pewna lista
 - wynik: łańcuch powstały przez połączenie elementów listy, rozdzielonych łańcuchem `str`

np:

```
print(', '.join(['1', '2', '3'])) → '1,2,3'
```

Podstawy programowania

funkcja list():

jak „list” → „lista”

- tworzy listę zawierającą kolejno wszystkie znaki z napisu
- `list(str)`:
 - argument: `str` – pewien napis
 - wynik: lista o długości `len(str)`, zawierająca znaki napisu
- np:
`print(list('abc')) → ['a', 'b', 'c']`

Podstawy programowania

metoda `ljust(szer)`:

jak „left justify” → „justuj w lewo”

- zwraca kopię łańcucha, wyjustowaną w lewo do szerokości `szer`
- `str.ljust(szer)`:
 - argument: `szer`: docelowa szerokość łańcucha
 - wynik: kopia łańcucha wyjustowana w lewo
- np:
`print(['+'+'Ala ma kota'.ljust(20)+'']) →`
`[Ala ma kota ]`

Podstawy programowania

metoda `ljust(szer,wypeł)`:

jak „left justify” → „justuj w lewo”

- zwraca kopię łańcucha, wyjustowaną w lewo do szerokości `szer` z wypełnieniem znakiem `wypeł`
- `str.ljust(szer)`:
 - argumenty: `szer`: docelowa szerokość łańcucha
`wypeł`: znak wypełniający (dokładnie jeden!)
 - wynik: kopia łańcucha wyjustowana w lewo
- np:
`print(['+'+'Ala ma kota'.ljust(20,'*')+'']) →`
`[Ala ma kota*****]`

Podstawy programowania

metoda lower():

jak „lower” → „małe (litery)”

- zwraca kopię łańcucha ze wszystkimi literami zamienionymi na małe
- `str.lower()`:
 - argumenty: żaden
 - wynik: kopia łańcucha z literami zmienionymi na małe
- np:
`print('ŁODZIA'.lower())` → `łodzią`

Podstawy programowania

metoda `lstrip()`:

jak „left strip” → „oddziel z lewej”

- zwraca kopię łańcucha z usuniętymi wiodącymi białymi znakami
- `str.lstrip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu wiodących białych znaków
- np:
`print('[ '+ x '.lstrip()+']')` → `[x ]`

Podstawy programowania

metoda `lstrip(zbiór)`:

jak „left strip” → „oddziel z lewej”

- zwraca kopię łańcucha z usuniętymi wiodącymi znakami ze zbioru `zbiór`
- `str.lstrip(zbiór)`:
 - argumenty: `zbiór`: łańcuch ze znakami do usunięcia
 - wynik: kopia łańcucha po usunięciu wiodących znaków
- np:
`print('www.zut.pl'.lstrip('w.')) → zut.pl`

Podstawy programowania

metoda `replace(stary,nowy)`:

jak „replace” → „zamień”

- zwraca kopię łańcucha z podłańcuchami `stary` zamienionymi na `nowy`
- `str.replace(stary,nowy)`:
 - argumenty: `stary`: łańcuch do zamiany
`nowy`: łańcuch zamieniający
 - wynik: kopia łańcucha po zamianie
- np:
`print('to nie to'.replace('to','ty'))` →
`ty nie ty`

Podstawy programowania

metoda `replace(stary,nowy,max)`:

jak „replace” → „zamień”

- zwraca kopię łańcucha z co najwyżej max podłańcuchami stary zamienionymi na nowy
- `str.replace(stary,nowy,max)`:
 - argumenty: stary: łańcuch do zamiany
nowy: łańcuch zamieniający
max: maksymalna liczba zamian
 - wynik: kopia łańcucha po zamianie
- np:
`print('to nie to'.replace('to','ty',1))` →
ty nie to

Podstawy programowania

metoda `rfind(podłańcuch)`:

jak „right find” → „znajdź od prawej”

- szuka podłańcucha w łańcuchu (jak `find()`), ale szuka od końca!)
- `str.rfind(podłańcuch)`:
 - argumenty: podłańcuch: szukany łańcuch
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to jest to'.rfind('to')) → 8`

Podstawy programowania

metoda `rfind(podłańcuch, start)`:

jak „right find” → „znajdź”

- szuka podłańcucha od końca łańcucha do pozycji `start`
- `str.rfind(podłańcuch, start)`:
 - argumenty: `podłańcuch`: szukany łańcuch
`start`: pozycja, do której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to jest tam'.rfind('to', 3)) → -1`

Podstawy programowania

metoda `rfind(podłańcuch,start,meta)`:

jak „right find” → „znajdź”

- szuka podłańcucha od końca łańcucha do pozycji start
począwszy od pozycji meta
- `str.find(podłańcuch,start,meta)`:
 - argumenty: podłańcuch: łańcuch
start: pozycja, do której należy szukać
koniec: pozycja, od której należy szukać
 - wynik: pozycja podłańcucha w łańcuchu albo -1
- np:
`print('to to to'.rfind('to',3,5)) → 3`

Podstawy programowania

metoda `rjust(szer)`:

jak „right justify” → „justuj w prawo”

- zwraca kopię łańcucha, wyjustowaną w prawo do szerokości `szer`
- `str.rjust(szer)`:
 - argument: `szer`: docelowa szerokość łańcucha
 - wynik: kopia łańcucha wyjustowana w prawo
- np:

```
print(['+'+'Ala ma kota'.rjust(20)+'']) →  
[  
    Ala ma kota]
```

Podstawy programowania

metoda `rstrip()`:

jak „right strip” → „oddziel z prawej”

- zwraca kopię łańcucha z usuniętymi końcowymi białymi znakami
- `str.rstrip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu kończących białych znaków
- np:
`print('['+' x '.rstrip()+']')` → `[ x]`

Podstawy programowania

metoda `rstrip(zbiór)`:

jak „right strip” → „oddziel z prawej”

- zwraca kopię łańcucha z usuniętymi kończącymi znakami ze zbioru `zbiór`
- `str.rstrip(zbiór)`:
 - argumenty: `zbiór`: łańcuch ze znakami do usunięcia
 - wynik: kopia łańcucha po usunięciu końcowych znaków
- np:
`print('www.zut.pl'.rstrip('l.p'))` → `www.zut`

Podstawy programowania

metoda split():

jak „split” → „rozdziel”

- zwraca listę podłańcuchów powstałą poprzez podzielenie danego łańcucha w miejscach, w których występują białe znaki
- `str.split()`:
 - argumenty: żaden
 - wynik: tablica podłańcuchów
- np:

```
print('ala ma kota'.split()) →  
['ala', 'ma', 'kota']
```

Podstawy programowania

metoda startswith(przedrostek):

jak „starts with” → „zaczyna się na”

- sprawdza, czy łańcuch zaczyna się ciągiem znaków przedrostek
- `str.startswith(przedrostek):`
 - argumenty: przedrostek: łańcuch
 - wynik: True / False
- np:
`print('Ala ma kota'.startswith('ta')) → False`

Podstawy programowania

metoda strip():

jak „strip” → „oddziel”

- zwraca kopię łańcucha z usuniętymi wiodącymi i końcowymi białymi znakami
- `str.strip()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po usunięciu wiodących i kończących białych znaków
- `np:`
`print(['+' x ' '.strip()+'']) → [x]`

Podstawy programowania

metoda swapcase():

jak „swap case” → „przerzuć rejestr liter”

- zwraca kopię łańcucha z małymi literami zamienionymi na wielkie i odwrotnie
- `str.swapcase()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po zamianie liter
małe $\Leftrightarrow$ wielkie
- np:
`print('Łódź'.swapcase())` → ŁÓDŹ

Podstawy programowania

metoda title():

jak „title” → „tytuł”

- zwraca kopię łańcucha z pierwszymi literami słów zamienionymi na wielkie i pozostałymi literami zamienionymi na małe
- `str.title()`:
 - argumenty: żaden
 - wynik: kopia łańcucha po „tytularyzacji”
- np:
`print('most NA rzece KWAI'.title())` →
Most Na Rzece Kwai

Podstawy programowania

metoda upper():

jak „upper” → „wielkie (litery)”

- zwraca kopię łańcucha ze wszystkimi literami zamienionymi na wielkie
- `str.upper()`:
 - argumenty: żaden
 - wynik: kopia łańcucha z literami zmienionymi na wielkie
- np:
`print('Łodzia'.upper())` → ŁODZIA

Podstawy programowania

Porównywanie łańcuchów

Podstawy programowania

Porównywanie łańcuchów:

- łańcuchy można porównywać ze sobą niemal tak samo jak liczby, przy użyciu tych samych operatorów:
`==` `!=` `>` `>=` `<` `<=`
- porównania typu większy-mniejszy zakładają, że do ustalenia relacji wykorzystuje się kody znaków ASCII/UNICODE, co może dawać zaskakujące rezultaty

Podstawy programowania

- **Uwaga:**

uporządkowanie liter w kodowaniu
ASCII/UNICODE jest dalekie od narzucających
się intuicji, dlatego należy zachować ostrożność!

Podstawy programowania

- bez niespodzianki:

```
print('a' < 'q')
```

True

Podstawy programowania

- niespodzianka!

```
print('b' > 'q')
```

False

Podstawy programowania

- **Zachowajmy spokój:**
- dzieje się tak dlatego, że litera 'ą' (leżąca poza ASCII, na wysokich kodach przypisanych w UNICODE), ma kod **większy** od wszystkich liter łacińskich
- na szczęście, nie ma to żadnego wpływu na sortowanie napisów (to odbywa się z zachowaniem reguł języka naturalnego)

Podstawy programowania

Porównywanie łańcuchów:

- `'ała' == 'ała'` → `True`
- `'Ała' == 'ała'` → `False`
- `'Oła' > 'Ała'` → `True`
- `'Kował' < 'Kowalski'` → `True`
- `'zona' > 'żona'` → `False`

Podstawy programowania

Porównywanie łańcuchów:

- `'10' == '010'` → `False`
- `'10' > '010'` → `True` (sic!)
- `'10' > '8'` → `False` (sic!)
- `'20' < '8'` → `True` (sic!)
- `'20' < '80'` → `True`

Podstawy programowania

Zapamiętaj:

- przy wykonywaniu porównań miej świadomość **typu** porównywanych danych
- w szczególności nie próbuj porównywać łańcuchów z liczbami – to może skończyć się **katastrofą**
- jedynymi dopuszczalnym porównaniami łańcucha z liczbą są `==` i `!=`
- pierwszy zawsze daje False, drugi zawsze True

Podstawy programowania

Porównywanie łańcuchów i liczb:

- `'10' == 10` → `False`
- `'10' != 10` → `True`
- `'10' == 1` → `False`
- `'10' != 1` → `True`
- `'10' > 10` → **BŁĄD**

Podstawy programowania

Sortowanie → funkcja sorted()

- sekwencja może być argumentem **funkcji** sorted()
- funkcja sorted() zwraca **posortowaną kopię** oryginalnej sekwencji
- sama sekwencja pozostaje **niezmieniona**

Podstawy programowania

- `sorted()` - przykład

```
sek1 = [5,4,3,2,1]  
sek2 = sorted(sek1)  
print(sek1)  
print(sek2)
```

```
[5, 4, 3, 2, 1]  
[1, 2, 3, 4, 5]
```

Podstawy programowania

Sortowanie → metoda sort()

- każda sekwencja ma **metodę** sort()
- metoda sort() **sortuje** oryginalną sekwencję
- metoda sort() **nie zwraca** wyniku

Podstawy programowania

- `sort()` - przykład

```
sek1 = [5,4,3,2,1]  
print(sek1)  
sek1.sort()  
print(sek1)
```

```
[5, 4, 3, 2, 1]  
[1, 2, 3, 4, 5]
```

Podstawy programowania

Zapamiętaj:

- sortowanie będzie przebiegać w zależności od **typu** sortowanych danych
- liczby będą sortowane **inaczej**, niż nawet bardzo podobnie wyglądające napisy

Podstawy programowania

- sortowanie liczb i napisów - przykład

```
print(sorted([11, 1, 8, 9, 90, 15]))  
print(sorted(['11', '1', '8', '9', '90', '15']))
```

```
[1, 8, 9, 11, 15, 90]  
['1', '11', '15', '8', '9', '90']
```

Podstawy programowania

Konwersja liczba → łańcuch

- liczbę można zamienić na reprezentujący ją łańcuch przy użyciu funkcji `str()`
- takie przekształcenie jest możliwe **zawsze**

<code>str(0)</code>	→	<code>'0'</code>
<code>str(-1)</code>	→	<code>'-1'</code>
<code>str(3.14)</code>	→	<code>'3.14'</code>

Podstawy programowania

Konwersja łańcuch → liczba

- łańcuch można zamienić na odpowiadającą mu liczbę używając znanych już funkcji `int()` oraz `float()`
- **uwaga** – takie przekształcenie nie zawsze jest możliwe (czemu?)