Laboratorium kryptograficzne dla licealistów 1

Projekt „Matematyka dla ciekawych świata”
Łukasz Mazurek

28.04.2016

1 Praca z Pythonem

Na zajęciach będziemy programować w języku Python w wersji 3. Pythona można używać na jeden z dwóch sposobów:

Zainstalowany interpreter Pythona Interpreter Pythona jest zainstalowany na komputerach w laboratorium. Możecie go również bezpłatnie pobrać ze strony `https://www.python.org/downloads/` i zainstalować na swoich domowych komputerach. Zwróćcie uwagę, aby zainstalować wersję o numerze rozpoczynającym się od 3 (np. 3.5.1), a nie starszą, ale wciąż używaną wersję 2. Wersje te różnią się tak znacząco, że programy, które będziemy pisać na zajęciach nie będą działać w starszej, drugiej wersji Pythona.

Interpreter Pythona online Istnieje wiele interpreterów Pythona online. Na zajęciach będziemy korzystali z interpretera znajdującego się na stronie `http://repl.it` . Interpretera tego będziecie mogli również używać na swoich domowych komputerach bez instalacji żadnego dodatkowego oprogramowania — wystarczy dowolna przeglądarka i podłączenie do internetu.

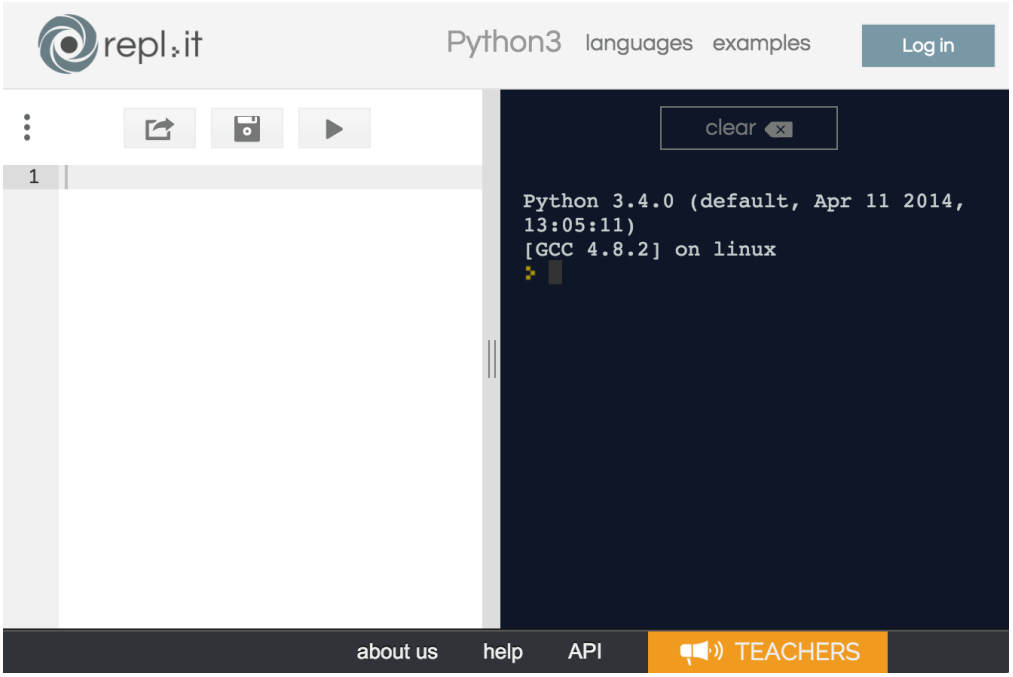
Innymi zasługującymi na uwagę interpreterami online są: `http://www.tutorialspoint.com/execute_python_online.php` i `http://ideone.com`

2 Praca z interaktywną konsolą Pythona `repl.it`

Uruchom system Windows (na Linuxach w laboratorium nie działa strona `http://repl.it`), otwórz przeglądarkę internetową, wejdź na stronę `http://repl.it` i wybierz język *Python 3*. Powinieneś zobaczyć dwa okienka: miejsce na kod (białe, z lewej) i *konsolę* (czarne, z prawej, zwane również *terminalem*), tak jak na rysunku 1.

Pierwszym sposobem pracy z Pythonem jest praca w interaktywnej konsoli, czyli praca w prawym okienku. W konsoli tej początkowo wypisane są pewne informacje (m. in. używana wersja Pythona) oraz znak zachęty `>`. Interpreter oczekuje, iż po tym znaku wpiszemy polecenie i naciśniemy Enter. Wynik polecenia zostanie wypisany w kolejnym wierszu (poprzedzony znakiem `=>`). Najprostszym sposobem użycia konsoli Pythona jest użycie jej jako kalkulatora — wpisujemy działanie do obliczenia, naciskamy Enter i w kolejnym wierszu otrzymujemy wynik działania. Przykład użycia konsoli Pythona jako kalkulatora znajduje się poniżej:

```
 > 2 + 2 * 2
=> 6
 > (2 + 2) * 2
=> 8
 > 2 ** 7
=> 128
 > 47 / 10
```



Rysunek 1: Widok interpretera Pythona dostępnego na stronie `http://repl.it`

```
=> 4.7
 > 47 // 10
=> 4
 > 47 % 10
=> 7
```

W powyższym przykładzie:

- Znak `**` oznacza podnoszenie do potęgi.
- Znak `/` oznacza dokładne dzielenie.
- Znak `//` oznacza dzielenie całkowite.
- Znak `%` oznacza branie reszty z dzielenia.

Podobnie jak w kalkulatorze możemy korzystać z *pamięci*, w Pythonie możemy zapisywać wartości w *zmiennych*:

```
 > x = 3
=> None
 > y = 4
=> None
 > x
=> 3
 > x**2 + y**2
=> 25
```

W pierwszej linijce następuje *przypisanie* wartości 3 do zmiennej `x`. Zwóć uwagę, że po wykonaniu tej komendy interpreter wypisuje `=> None`. Oznacza to, że linijka ta nie zwraca żadnego wyniku. Nie przejmujemy się tym. Ważne, że wartość 3 została zapamiętana w zmiennej `x` i od tej pory możemy z niej korzystać.

Zmienne mogą również jako wartości przyjmować litery, słowa, a nawet całe zdania:

```
 > a = 'Ala'
 > b = "ma"
```

```
 > c = 'kota i wiele innych zwierząt'
 > a[0]
=> 'A'
 > c[1]
=> 'o'
 > a + b
=> 'Alama'
 > a + b + c
=> 'Alamakota i wiele innych zwierząt'
```

Zwróć uwagę na następujące rzeczy:

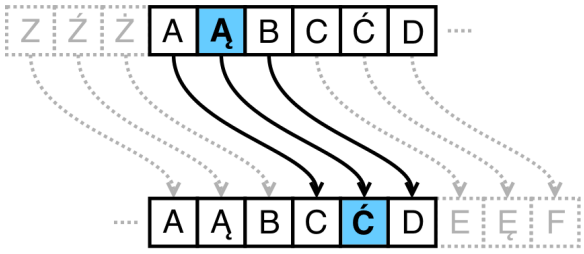
- Napisy muszą być otoczone pojedynczymi apostrofami lub podwójnym cudzysłowami (nie ma znaczenia, którą wersję wybierzemy).
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery słowa (numeracja rozpoczyna się od 0).
- Przy użyciu znaku dodawania możemy sklejać (*konkatenować*) napisy.

2.1 Szyfr Cezara

Uwaga *We wszystkich zadaniach dotyczących szyfrowania (o ile nie powiedziano inaczej) zajmujemy się słowami składającymi się wyłącznie z wielkich liter 32-literowego polskiego alfabetu (AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ).*

Szyfr Cezara polega na zastąpieniu każdej litery tekstu jawnego literą występującą w alfabecie o 3 pozycje później — literę A szyfrujemy jako C, Ą szyfrujemy jako Ć, itd. Gdy „skończy nam się alfabet", zaczynamy liczyć od początku — literę Z szyfrujemy jako A, Ź jako Ą, a Ż jako B. Czyli szyfrogramem dla tekstu jawnego CEZAR jest słowo EGACT, a szyfrogram DCÓCÓ po odszyfrowaniu daje tekst jawny BANAN.

Zadanie 1 *Odszyfruj przy użyciu kartki i długopisu (nie wspomagając się komputerem) tekst jawny, który po zaszyfrowaniu szyfrem Cezara daje szyfrogram ŚŻAC.*



W oryginalnym szyfrze Cezara następowało przesunięcie o 3 pozycje w alfabecie. Możemy zajmować się również *szyfrem Cezara z kluczem k*, w którym szyfrogram otrzymuje się przesuwając każdą literę tekstu o k pozycji w alfabecie. Np. dla $k = 5$ szyfr przeprowadza A na D, Ą na E, itd. O ile dla $k = 3$ można stosunkowo szybko odszyfrować pojedynczą literę w pamięci (cofając się o 3 litery w alfabecie), o tyle już np. dla $k = 10$ operacja ta staje się bardziej czasochłonna. Spróbujmy więc pomóc sobie, wykorzystując Pythona.

Alfabet i polecenie `index()` Aby ułatwić sobie szyfrowanie, stwórzmy zmienną `alfabet`, która będzie zawierała słowo składające się z kolejnych liter alfabetu:

```
 > alfabet = 'AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ'
```

Aby się upewnić, czy dobrze przepisaliśmy alfabet, możemy sprawdzić jego długość przy użyciu polecenia **len**`()`:

```
 > len(alfabet)
=> 32
```

Jak wiemy, do kolejnych liter słowa odwołujemy się przy użyciu nawiasu kwadratowego. W ten sam sposób możemy uzyskać poszczególne litery alfabetu:

```
 > alfabet[0]
=> 'A'
 > alfabet[31]
=> 'Ż'
```

Stosując polecenie `index()` uzyskujemy operację odwrotną — podając literę, możemy poznać jej pozycję w alfabecie:

```
 > alfabet.index('Ą')
=> 1
 > alfabet.index('Y')
=> 28
```

Łącząc dwie powyższe operacje uzyskujemy narzędzie do szyfrowania szyfrem Cezara!

```
 > pozycja = alfabet.index('B')
 > alfabet[pozycja + 3]
=> 'D'
```

Albo prościej, w jednej linijce:

```
 > alfabet[alfabet.index('B') + 3]
=> 'D'
```

Rozszyfrowywanie szyfru Cezara Spróbujmy teraz rozszyfrować wiadomość zaszyfrowaną szyfrem Cezara z kluczem 10. Skoro przy szyfrowaniu do pozycji w alfabecie każdej litery zostało dodane 10, to aby rozszyfrować, należy od pozycji każdej litery odjąć 10:

```
 > alfabet[alfabet.index('J') - 10]
=> 'B'
 > alfabet[alfabet.index('A') - 10]
Traceback (most recent call last):
  File "python", line 1, in <module>
IndexError: string index out of range
```

Widzimy, że procedura ta działa, o ile nie wyjdziemy poza zakres alfabetu AĄB...ZŹŻ. Przy próbie cofnięcia o 10 litery A wyszliśmy poza ten zakres i Python zasygnalizował błąd. W takim przypadku, aby uzyskać odszyfrowaną literę, zamiast odejmować od pozycji 10, możemy do niej dodać 22 (efekt będzie ten sam, ponieważ łącznie w alfabecie są 32 litery):

```
 > alfabet[alfabet.index('A') + 22]
=> 'R'
```

Zadanie 2 *Korzystając z wymienionych wyżej metod, odszyfruj wiadomość, która po zaszyfrowaniu szyfrem Cezara z kluczem 10 dała szyfrogram JAĘĆKÓZUŹDKH.*

2.2 Pętla for

Aby rozwiązać ostatnie zadanie musieliśmy 12 razy (albo więcej, jeśli się pomyliliśmy) wykonać bardzo podobną komendę. Do wykonywania wielokrotnie tego samego (lub podobnego) kodu służą pętle. Najprostszym rodzajem pętli jest pętla **for**, która dla danej listy i operacji do wykonania wykonuje tę operację po

kolei na każdym elemencie listy. Powiedzmy, że chcemy obliczyć kwadraty liczb: 1, 2, 3, 4, 5. Zgodnie z naszą dotychczasową wiedzą, zrobilibyśmy to tak:

```
 > 1 * 1
=> 1
 > 2 * 2
=> 4
 > 3 * 3
=> 9
 > 4 * 4
=> 16
 > 5 * 5
=> 25
```

Do zautomatyzowania tego procesu służy pętla **for** w następującej postaci:

```
 > for x in [1, 2, 3, 4, 5]:
..     print(x * x)
..
1
4
9
16
25
```

Zwróć uwagę na kilka rzeczy:

- Na końcu pierwszej linijki jest dwukropek.
- Gdy naciśniemy Enter po zakończeniu pierwszej linijki, znak zachęty zmieni się z > na .., co oznacza to, że jesteśmy w trakcie pisania polecenia wielolinijkowego.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji. Jeśli zapomnimy o wcięciu, interpreter zgłosi błąd i całą komendę wielolinijkową będziemy musieli pisać od początku (można pomóc sobie, naciskając strzałkę w górę i przeglądając stare komendy).
- Po wpisaniu drugiej linijki i naciśnięciu Entera pojawi się kolejny znak zachęty .., co oznacza, że interpreter czeka na ciąg dalszy polecenia wielolinijkowego. Jeśli nie chcemy pisać dalszego ciągu, w tym momencie musimy jeszcze raz nacisnąć Enter.

3 Pisanie i uruchamianie kodu programu

Do tej pory korzystaliśmy z Pythona używając interaktywnej konsoli (czyli prawego okienka). Jest to całkiem wygodne narzędzie, jeśli wykonujemy tylko jednolinijkowe polecenia, jednak pisanie dłuższych fragmentów kodu w tej konsoli staje się już bardzo niewygodne. Drugą metodą korzystania z Pythona jest pisanie kodu programu (skryptu) w pliku tekstowym (w lewym okienku) i uruchamianie tego kodu w konsoli (w prawym okienku).

Spróbuj teraz napisać w lewym okienku kod tej samej pętli for (pamiętając o dwukropku na końcu pierwszej i wcięciu drugiej linijki):

```
for x in [1, 2, 3, 4, 5]:
    print(x * x)
```

i wcisnąć przycisk „run” znajdujący się na górze tego okienka. Po chwili w konsoli po prawej powinien pojawić się wynik działania naszego programu:

```
1
4
9
16
25
```

Od tej pory właśnie w taki sposób będziemy pisać i uruchamiać wszystkie nasze programy. Ilekroć w niniejszych materiałach pojawią się dwie ramki, jedna obok drugiej, w lewej ramce znajdował się będzie kod programu, a w prawej efekt jego działania wyświetlony w konsoli (pomijamy nic nie wnoszące `=> None` na końcu):

```
for x in [1, 2, 3, 4, 5]:
    print(x * x)
```

```
1
4
9
16
25
```

3.1 Lista słów

W powyższym przykładzie pętla **for** przechodziła po kolejnych elementach listy `[1, 2, 3, 4, 5]` i w każdym kroku przypisywała kolejną liczbę z listy na zmienną `x`, a następnie wykonywała pewne polecenie związane z tą zmienną. Oczywiście zmienna, na którą przypisywane są kolejne elementy listy, nie musi się nazywać `x`, a elementy listy nie muszą być liczbami:

```
for slowo in ['Ala', 'ma', 'kota']:
    print(2 * (slowo + 'hh'), end = ' ')
```

```
AlahhAlahh mahhmahh kotahhkotahh
```

Na powyższym przykładzie widzimy, że:

- Pętla przechodzi po liście słów.
- Zmienna `slowo` przechowuje napisy (mówimy, że zmienna ta jest typu *string*).
- Napisy możemy konkatenować (sklejać) przy użyciu dodawania oraz powtarzać przy użyciu mnożenia przez liczbę.
- Używając dodatkowego argumentu `end = ...` możemy wymusić wypisanie spacji (lub dowolnego innego ciągu znaków) zamiast przejścia do nowej linii po końcu wypisywanego wyrażenia.

3.2 Słowo = lista liter

W poprzednim rozdziale zajmowaliśmy się *listą słów*. Okazuje się, że pojedyncze słowo również jest listą, a dokładniej listą liter. Wiemy już jak sprawdzać długość słowa i jak odwoływać się do poszczególnych jego liter:

```
 > slowo = 'Python'
 > len(slowo)
=> 6
 > slowo[0]
=> 'P'
 > slowo[5]
=> 'n'
```

Litery można numerować od końca (-1 to ostatnia litera, -2 przedostatania, itd.):

```
 > slowo[-1]
=> 'n'
 > slowo[-4]
=> 't'
```

Można też wyciąć podsłowo:

```
 > slowo[2:5]
=> 'tho'
 > slowo[3:]
=> 'hon'
 > slowo[:3]
=> 'Pyt'
```

W powyższym przykładzie pierwsza komenda zwraca podsłowo od znaku nr 2 (**włącznie**) do znaku nr 5 (**wyłącznie**), druga podsłowo **od** znaku nr 3 (**włącznie**), a trzecia podsłowo **do** znaku nr 3 (**wyłącznie**).

Do zapamiętania: ***Wszystkie przedziały w Pythonie są domknięte z lewej strony i otwarte z prawej strony, tzn. zawierają swój lewy koniec i nie zawierają swojego prawego końca.***

Skoro słowo jest listą, to znaczy, że można po nim przejść przy użyciu pętli **for**:

```
for l in 'Kula':
    print('litera', end = ' ')
    print(l)
```

```
litera K
litera u
litera l
litera a
```

Zwróć uwagę, że wewnątrz pętli for może znajdować się więcej niż jedno polecenie. Trzeba tylko pamiętać, aby wszystkie były poprzedzone takim samym wcięciem.

Zadanie 3 *Przy użyciu pętli* **for** *odszyfruj słowo, które po zaszyfrowaniu szyfrem Cezara o kluczu 3 daje szyfrogram DCŃŹREKŹCNUYŹR. Zauważ, że w tym przypadku nie musimy się martwić „wychodzeniem liter poza alfabet".*

3.3 Lista kolejnych liczb naturalnych

Często potrzebujemy, aby pętla przeszła po liście kolejnych liczb naturalnych. W tym celu używamy polecenia **range**():

```
for x in range(7):
    print(x, end = ', ')
```

```
0, 1, 2, 3, 4, 5, 6,
```

```
for x in range(5, 10):
    print(x, end = ', ')
```

```
5, 6, 7, 8, 9,
```

```
for x in range(10, 20, 3):
    print(x, end = ', ')
```

```
10, 13, 16, 19,
```

Na powyższych przykładach widzimy, że polecenie **range**() występuje w trzech wersjach:

- **range**(kon) generuje listę kolejnych liczb od 0 (włącznie) do kon (wyłącznie).
- **range**(pocz, kon) generuje listę kolejnych liczb od pocz (włącznie) do kon (wyłącznie).
- **range**(pocz, kon, krok) generuje listę liczb od pocz (włącznie) do kon (wyłącznie), przeskakując w każdym kroku o krok.

4 Zadania dodatkowe

Zadanie 4 *Napisz pętlę, która dla danej listy słów* `lista` *utworzy akronim składający się z pierwszych liter każdego słowa. Np. dla listy* `['Matematyka', 'Informatyka', 'Mechanika']` *wypisze słowo* `MIM`.

Zadanie 5 *Napisz pętlę, która dla danego słowa* `slowo` *składającego się wyłącznie z małych liter alfabetu łacińskiego, wypisze je ze wszystkimi literami zamienionymi na wielkie. Np. dla* `slowo = 'laboratorium'` *program powinien wypisać*

```
LABORATORIUM
```

Zadanie 6 *Oblicz sumę* $1^2 + 2^2 + 3^2 + \ldots + 99^2 + 100^2$.

5 Praca domowa nr 1

Rozwiązania zadań domowych należy przesłać do czwartku 5 maja do godz. 16^{59} na adres `licealisci.pracownia@icm.edu.pl` wpisując jako temat wiadomości `Lx PD1`, gdzie `x` to numer grupy, np. `L3 PD1` dla grupy L3, itd.

Zadanie domowe 1 (1 pkt) *Napisz pętlę, która wypisze wszystkie dodatnie potęgi dwójki mniejsze od tysiąca (nie chodzi o tysięczną potęgę dwójki, tylko wypisane liczby mają być mniejsze niż 1000). Kolejne liczby powinny być wypisane w jednym wierszu i porozdzielane pojedynczymi spacjami.*

Zadanie domowe 2 (2 pkt) *Napisz pętle, która dla danej listy słów* `lista` *wypisze w kolejnych wierszach ich skróty w postaci* `<pierwsza litera>-<ostatnia litera> (<dlugosc slowa>)`. *Np. dla listy* `lista = ['Interdyscyplinarne', 'Centrum', 'Modelowania']` *powinna wypisać:*

```
I-e (18)
C-m (7)
M-a (11)
```

Wskazówka: wynik funkcji **len()** *mierzącej długość słowa jest liczbą. Do rozwiązania tego zadania może Ci się przydać konwersja tej liczby na słowo (aby dało się ją skleić z innymi słowami). Do tego służy polecenie* **str()**:

```
 > len('dudy')
=> 4
 > str(len('dudy'))
=> '4'
```

Zadanie domowe 3 (3 pkt) *Napisz pętlę, która zaszyfruje dane słowo zapisane w zmiennej* `slowo` *szyfrem Cezara o kluczu* `k`. *Pętla powinna działać poprawnie dla każdego słowa składającego się z wielkich liter polskiego alfabetu (AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ) i dla każdego klucza* $k \in \{0, 1, \ldots, 31\}$. *W tym zadaniu trzeba już radzić sobie z „wychodzeniem poza zakres alfabetu".*

Pewne słowo zostało zaszyfrowane szyfrem Cezara o kluczu 15 i otrzymano szyfrogram: GEŁFIESŁYŃWBABHŁAI. Użyj swojego programu, aby zrekonstruować tekst, który został zaszyfrowany.

Wskazówka: problem wychodzenia poza zakres alfabetu możemy sprowadzić do problemu brania reszty z dzielenia przez 32. Ponumerujmy kolejne litery alfabetu indeksami od 0 do 31. Jeśli w wyniku szyfrowania otrzymamy np. indeks 35, to wiemy, że w rzeczywistości, chodzi o literę o indeksie 3, czyli o indeksie równym reszcie z dzielenia 35 przez 32. Możemy zatem najpierw standardowo obliczyć indeks zaszyfrowanej litery (być może większy niż 31), a następnie obliczyć jego resztę z dzielenia przez 32 (w Pythonie resztę z dzielenia liczymy przy użyciu znaku procenta) i uzyskamy w ten sposób prawdziwy indeks szukanej litery w alfabecie.

Inne podejście do tego problemu: można „skleić" ze sobą dwa alfabety: AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻAĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ