Tytuł rozdziału

Programowanie funkcyjne

Python – rozpoczynamy!

- Python jest językiem programowania, który według jego twórców ma na celu skutecznie połączyć „zasługującą na uwagę moc z bardzo przejrzystą składnią”.
  - W rzeczy samej, już w bardzo nieodległym czasie, będziemy w stanie pisać programy rozwiązujące ciekawe i pożyteczne zadania.
- Licząc na to, że prezentowany dalej materiał wykorzystujący język Python będzie interesującą lekturą, jednocześnie należy podkreślić, że nie ma lepszego sposobu opanowania przekazywanych treści jak samodzielne ćwiczenie.
  - Dlatego należy tak często jak tylko zachodzi potrzeba odrywać się od czytania aby wypróbować na komputerze prezentowane rozwiązania.
  - Ponadto ten kurs jest stosunkowo krótki i konkretny. Aby należycie przyswoić materiał, ważne jest aby ćwiczyć i wykonywać zadane prace.
- Zaczynamy! Po uruchomieniu Python przedstawia się znakiem „zachęty”, który, na przykład, może mieć postać:

```
>>>
```

Stanowi to zaproszenie do wpisywania rozmaitych rzeczy.

Użycie Pythona jako kalkulatora (1/2)

☞ Na początek wpisujmy wyłącznie wyrażenia arytmetyczne – co w istocie oznacza wykorzystanie systemu jako kalkulatora.

- Przykładem może być wpisanie 3+5 i uzyskanie wyniku.

```
>>> 3 + 5
8
```

- Dalej możemy wykonywać operacje bardziej złożone, jak:

```
>>> (3 + 5) * 2 - 1
15
```

☞ Zauważmy, że nawiasów użyto do kontrolowania kolejności wykonywania operacji.

- Tradycyjnie mnożenie i dzielenie mają wyższy priorytet niż dodawanie i odejmowanie, co oznacza, że Python wykonuje najpierw mnożenia i dzielenia a po nich dodawania i odejmowania. Zatem bez nawiasów otrzymalibyśmy:

```
>>> 3 + 5 * 2 - 1
12
```


Użycie Pythona jako kalkulatora (2/2)

☞ Można rozważyć kolejne przykłady obliczeń arytmetycznych w Pythonie:

```
>>> 3 + 5
8
>>> (3 + 5) * 2 - 1
15
>>> 3 + 5 * 2 - 1
12
>>> 6 / 2
3.0
```

```
>>> 2 ** 5
32
>>> 10 ** 3
1000
>>> 52 % 10
2
```

- Możemy domyślić się znaczenia symboli /, *, ** oraz %. W szczególności wyrażenie 52 % 10 oznacza resztę z dzielenia 52 przez 10 — określaną także jako "52 mod 10". Symbole arytmetyczne +, -, /, *, ** oraz % nazywamy operatorami.
- Jeśli w Python 3 chcemy wykonać dzielenie całkowite (oznaczające podzielenie dwóch liczb całkowitych o uzyskanie w wyniku liczby całkowitej), musimy użyć specjalnego znaku operacji dzielenia całkowitego //:

```
>>> 11 // 2
5
```

Nadawanie nazw (1/2)

☞ Python umożliwia nadawanie nazw wartościom stanowiącym wyniki obliczeń.

- Oto przykład:

```
>>> pi = 3.1415926
>>> pi * (10 ** 2)
314.15926
```

- W linii pierwszej nadaliśmy wartość dla pi jako równą 3.1415926. W drugiej linii użyliśmy tej wartości do obliczenia pola koła o promieniu 10.
- Informatycy określają nazwy takie jak "pi" jako zmienne ponieważ można nadawać im dowolne wartości. I rzeczywiście, jeśli zechcemy, zmiennej „pi" możemy nadawać nowe wartości, nawet tak szalone jak np. 42.
- Warto tutaj uświadomić sobie, że pojęcie „zmiennej" w informatyce różni się od zmiennej w matematyce. Żaden matematyk nie określi liczby π jako zmiennej!

Nadawanie nazw (2/2)

☞ Zauważmy, że znak "=" jest używany do nadawania wartości zmiennej. Po lewej stronie znaku równości znajduje się nazwa zmiennej. Po prawej umieszcza się wyrażenie, którego wartość jest wyznaczana i nadawana zmiennej.

- Na przykład możemy zrobić coś takiego:

```
>>> pi = 3.1415926
>>> area = pi * (10 ** 2)
>>> area
314.15926
```

- W pierwszej linii przykładu definiujemy zmienną o nazwie pi. W linii drugiej jest obliczana wartość wyrażenia pi * (10 ** 2) (wynosi ona 314.15926) a następnie wartość ta jest nadawana innej zmiennej o nazwie area. Następnie po wprowadzeniu area Python wyświetla obliczoną wartość.
- Należy zwrócić uwagę, że znak równości reprezentuje akcję, która prowadzi do takiego zmodyfikowania zmiennej po lewej stronie (w tym przypadku pi oraz area), że przyjmuje ona nową wartość. Wartość ta może zostać zmieniona później. Ważne jest, aby odróżniać taką rolę znaku równości od jego znaczenia w matematyce, gdzie równość wskazuje, że lewa i prawa strona są zawsze takie same. Jak widzimy w informatyce jest inaczej!

Jak dobierać nazwy?

- Python nie jest specjalnie wymagający w kwestii nadawania nazw zmiennym.
  - Na przykład nazwy Joe czy też Joe42 są poprawne. Jednak nadanie zmiennej nazwy Joe+Sally nie jest dopuszczalne. Nietrudno jest domyślić się przyczyny: kiedy Python napotka Joe+Sally to uzna, że chcemy dodać do siebie wartości dwóch zmiennych Joe i Sally.
  - Oprócz tego w języku Python istnieją wbudowane słowa specjalne, których nie można użyć jako nazw zmiennych. Próba takiego użycia zakończy się komunikatem o błędzie.
  - Zamiast wypisywać tutaj słowa których nie mogą w Pythonie być nazwami zmiennych, warto po prostu mieć świadomość, że komunikat o błędzie w przypadku asygnowania zmiennej jest prawdopodobnie efektem natrafienia na jedną z nielicznych zabronionych nazw.
- Ważne jest także używanie takich nazw, które dobrze opisują znaczenie zmiennych i pomagają czytającemu zrozumieć nasz program.
  - Na przykład, jeśli pewna zmienna ma przechowywać wartość pola koła, to nazwanie jej area (lub nawet areaOfCircle) będzie o wiele lepszym wyborem niż użycie czegoś w rodzaju x42b lub harriet.

Od liczb do łańcuchów znakowych

- Jednym z centralnych tematów w informatyce są dane. Dotychczas operowaliśmy tylko jednym rodzajem danych: liczbami.
  - Wszystko pięknie, ale liczby nie są jedynym przydatnym rodzajem danych.
  - Nadszedł czas, aby przedstawić kilka innych typów danych, które są ważne przy rozwiązywaniu problemów w języku Python.
- Załóżmy, że chcemy porównywać ze sobą łańcuchy znakowe.
  - Łańcuchem jest dowolny ciąg znaków umieszczony w apostrofach.
  - Python pozwala umieszczać łańcuchy zarówno podwójnych jak i w pojedynczych apostrofach.
  - Jednakże Python wyświetla łańcuchy w pojedynczych apostrofach, niezależnie od tego, czy użyliśmy pojedynczych czy podwójnych. Oto przykłady:

```
>>> name1 = "Ben"
>>> name2 = 'Jerry'
>>> name1
'Ben'
>>> name2
'Jerry'
```

Nazwy name1 i name2 odnoszą się do zdefiniowanych przez nas zmiennych. Spośród wielu operacji, które mogą być wykonane na łańcuchach przedstawione zostanie tylko kilka najważniejszych. W przykładach zakładamy, że zmiennym name1 i name2 nadano wartości w pokazany sposób.

Krótko na temat długości łańcucha znaków

☞ Po pierwsze jest możliwe wyznaczenie długości łańcucha za pomocą funkcji len:

```
>>> len(name1)
3
>>> len('I love spam!')
12
```

- W pierwszym przykładzie name1 to łańcuch znaków 'Ben', a długość tego łańcucha wynosi 3.
- W przykładzie drugim łańcuch zawiera dwie spacje (spacja jest znakiem jak każdy inny i jest liczona do długości) dlatego całkowita długość jest równa 12.

Indeksy

☞ Inną operacją możliwą do wykonania na łańcuchach jest odnalezienie znaku umieszczonego na dowolnie zadanej pozycji (o dowolnym indeksie).

- W niektórych językach programowania, w tym także w Pythonie, pierwszy znak w łańcuchu ma indeks 0. Dlatego:

```
>>> name1[0]
'B'
>>> name1[1]
'e'
>>> name1[2]
'n'
>>> name1[3]
IndexError: string index out of range
```

- Zauważmy, że pomimo tego, iż zmienna name1 , której wartością jest 'Ben', ma długość 3, to poszczególne znaki mają indeksy 0, 1 i 2 ze względu na specyficzny sposób odliczania w języku Python. Nie ma znaku z indeksem 3, co powoduje, że pojawia się komunikat o błędzie. Generalnie informatycy mają skłonność do odliczania raczej od 0 niż od 1 .

Plasterkowanie (*slicing*) (1/2)

☞ Python umożliwia dostęp do fragmentów łańcucha poprzez użycie specjalnego zapisu określanego jako „plasterkowanie” (*slicing notation*).

- Na początek kilka przykładów:

```
>>> bestFood = 'spam burrito'
>>> bestFood[0:3]
'spa'
>>> bestFood[0:4]
'spam'
```

- Najpierw została zdefiniowana zmienna bestFood , a jej wartością stał się łańcuch‘spam burrito’.
- Zapis bestFood[0:3] służy do tego, aby przekazać systemowi Python, że ma dostarczyć część – lub „plasterek” - tego łańcucha począwszy od indeksu 0 i dalej, ale bez włączania znaku z indeksem 3.
- W ten sposób dostajemy fragment łańcucha obejmujący znaki z indeksami 0, 1 oraz 2, którymi są trzy litery s, p, a — tworzące łańcuch spa.
- Może się wydać dziwnym, że w przyciętym łańcuchu nie znalazł się znak z ostatnim indeksem, przez co właściwie nie dostajemy znaku z indeksem 3, kiedy prosimy o plasterek bestFood[0:3]. Okazuje się, że są uzasadnione powody dla których twórcy języka Python tak zdecydowali, a potwierdzające to przykłady odkładamy na później.

Plasterkowanie (*slicing*) (2/2)

☞ Plasterek nie musi rozpoczynać się od indeksu 0.

- Na przykład:

```
>>> bestFood[2:6]
'am b'
```

Otrzymujemy plasterek wycięty z łańcucha 'spam burrito' począwszy od indeksu 2 i dalej w górę aż do indeksu 6, ale bez włączania znaku z indeksem 6.

- Możemy także wykonywać operacje, jak w tym przykładzie:

```
>>> bestFood[1:]
'pam burrito'
```

Jeśli opuścimy liczbę po dwukropku, Python zakłada, że mówimy „idź do końca”. A zatem w przykładzie zadysponowano: „dostarcz plasterek , który rozpoczyna się od indeksu 1 i ciągnie się do samego końca”.

- I jeszcze takie:

```
>>> bestFood[:4]
'spam'
```

Ponieważ nic nie było przed dwukropkiem, przyjęto, że jest tam 0. A zatem użyty zapis oznacza to samo co bestFood[0:4].

Listy (1/2)

☞ Dotychczas przyjrzeliśmy się dwóm różnym typom danych: liczbom i łańcuchom znaków. Czasem przydatnym okazuje się „spakowanie” grupy liczb lub łańcuchów w jedną strukturę.

- Python operuje także innym typem danych nazywanym listą, która to umożliwia. Oto przykłady:

```
>>> oddNumbers = [1 , 3 , 5 , 7 , 9 , 11]
>>> friends = [ 'rachel' , 'monica' , 'phoebe' , 'joey' , 'ross']
```

- W pierwszej linii oddNumbers jest zmienną, którą desygnowano jako listę złożoną z sześciu liczb nieparzystych.
- W drugim poleceniu friends jest zmienną, której nadano wartość listy złożonej z pięciu łańcuchów.
- Po wpisaniu oddNumbers lub friends, Python pokaże ich aktualne wartości.
- Zauważmy, że lista rozpoczyna się nawiasem otwierającym [, kończy nawiasem zamykającym], a każdy element wewnątrz listy jest oddzielony od następnego przecinkiem.

Listy (2/2)

- Przyjrzyjmy się temu:

```
>>> stuff = [2 , 'hello' , 2.718]
```

Należy zauważyć, że stuff jest zmienną, którą desygnowano jako listę zawierającą dwie liczby i łańcuch znaków. Język Python dopuszcza aby wewnątrz tej samej listy znajdowały się obiekty różnych typów!

- W istocie, stuff może zawierać w sobie nawet inne listy, jak pokazuje ten przykład:

```
>>> stuff = [2 , 'hello' , 2.718 , [1 , 2 , 3]]
```

Taka zdolność współistnienia wielu typów danych w ramach jednej listy jest określana jako polimorfizm (co oznacza „wiele typów”) będący wspólną cechą języków programowania funkcyjnego.

Praca z listami (1/3)

☞ Prawie każda operacja działająca na łańcuchach znakowych działa także na listach.

- Na przykład można wyznaczyć długość listy w identyczny sposób, jak długość

```
>>> len(stuff)
4
```

Warto zauważyć, że lista stuff w rzeczywistości zawiera tylko cztery elementy: liczbę 2, łańcuch znaków 'hello', liczbę 2.718 oraz listę [1, 2, 3].

- Indeksy oraz plasterki także działają dla list analogicznie jak dla łańcuchów:

```
>>> stuff [0]
2
>>> stuff [1]
'hello'
>>> stuff [2:4]
[2.718, [1, 2, 3]]
```


Praca z listami (2/3)

- Tak samo jak łańcuchy znakowe, listy mogą być dodawane (składane) oraz mnożone.
- Dodawanie dwóch list prowadzi do utworzenia nowej listy zawierającej wszystkie elementy z pierwszej listy, po których następują wszystkie elementy z drugiej listy. Operacja taka jest nazywana konkatenacją list i jest podobna do konkatenacji

```
>>> mylist = [1, 2, 3]
>>> mylist + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> mylist * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Praca z listami (3/3)

☞ Warto jeszcze zauważyć, że operacja taka jak:

```
>>> mylist + [4 , 5 , 6]
```

w rzeczywistości nie zmienia mylist. Zamiast tego zwraca nową listę która jest wynikiem konkatenacji mylist oraz listy[4, 5, 6].

- Można to sprawdzić wpisując mylist po znaku „zachęty" i upewniając się, że lista mylist pozostaje taka sama:

```
>>> mylist + [4 , 5 , 6]
[1, 2, 3, 4, 5, 6]
>>> mylist
[1, 2, 3]
```


Funkcje

☞ W matematyce funkcję można wyobrazić sobie jako „skrzynkę" która pobiera pewną daną jako wejście lub argument a następnie zwraca pewną daną jako wyjście, które nazywamy wynikiem działania funkcji, wartością zwracaną lub po prostu wartością.

- Na przykład, funkcja $f(x)=2x$ ma argument nazwany literą x, a dla każdej wartości, którą „wetkniemy" do x otrzymamy wartość, która jest dwukrotnie większa.

☞ Python także umożliwia definiowanie funkcji, i , tak jak w matematyce, funkcje te pobierają dane jako argumenty, przetwarzają je w pewien sposób, po czym zwracają dane stanowiące wynik.

- Oto funkcja zdefiniowana w języku Python o nazwie f, pobierająca argument nazwany jako x oraz zwracająca wynik, który jest wartością dwa razy większą od x:

```
>>> def f ( x ):
    return 2 * x
```

Składnia definicji funkcji

- W języku Python, do definiowania funkcji wykorzystuje się słowo specjalne def, które oznacza: „właśnie def (-iniuję) funkcję”.
  - Potem podaje się nazwę, którą wybrano dla funkcji; w naszym przypadku jest to f.
  - Dalej następują argumenty funkcji podane w nawiasach (tak samo jak w definicji funkcji w matematyce!), a po nich dwukropek (“:”).
  - Następnie przechodzimy do nowej linii „wciętej” o kilka spacji i rozpoczynamy zapis treści funkcji. W rozważanym przypadku funkcja oblicza wartość 2*x a następnie ją zwraca.
  - Słowo return poleca systemowi Python, aby wyliczona wartość została zwrócona jako wynik działania funkcji.
  - Należy pamiętać o tym, aby linia (lub linie), które następują po linii pierwszej były zapisane z „wcięciem”. Jest to w Pythonie bezwzględny wymóg.

Wywołanie funkcji w interpreterze Pythona

- Możemy teraz uruchomić funkcję w interpreterze języka Python następująco:

```
>>> f (10)
20
```

```
>>> f (-1)
-2
```

  - Zapis f(10) nazywa się wywołaniem funkcji ponieważ, używając przenośni, „dzwonimy do funkcji” f załączając argument 10 z prośbą o udzielenie odpowiedzi, tj. dostarczenie obliczonej wartości.
  - Po wywołaniu f, Python bierze wartość podaną w nawiasach i nadaje tę wartość zmiennej x.
  - Następnie wykonywane jest polecenie lub polecenia składające się na funkcję w zadanym porządku, aż do ich wyczerpania.
  - W tym przypadku funkcja zawiera tylko jedno polecenie, które podwaja wartość x i zwraca wynik w miejsce, skąd nastąpiło wywołanie.

Wiersze dokumentujące (*docstrings*)

- W istocie rzeczy funkcja jest programem komputerowym. A zatem, właśnie napisaliśmy nasz pierwszy program!
  - Zapewne program, który podwaja wartość zadanej liczby nie jest tym, czym chcielibyśmy chwalić się przed przyjaciółmi i rodziną. Niemniej przybliżamy się do pisania pewnych ciekawych programów.
  - W miarę jak programy stają się bardziej interesujące, ważne jest aby ich użytkownicy byli w stanie w miarę szybko zrozumieć do czego te programy służą.
  - Dlatego każda funkcja, która od tej pory będzie przez nas napisana będzie rozpoczynać się od „wierszy dokumentujących" (*docstring - documentation string*). Jeśli rozważaną wcześniej funkcję f podwajającą wartość liczby uzupełnić o taki wiersz, to może ona wyglądać następująco:

```
def f ( x ):
    ''' Takes a number x as an argument and returns 2*x.'''
    return 2 * x
```

- Łańcuch dokumentujący rozpoczynają i kończą trzy pojedyncze apostrofy.
  - Jeśli teraz wprowadzimy help(f), to łańcuch dokumentujący jest wyświetlany.
  - Łańcuch dokumentujący można umieścić zawsze, w każdej funkcji.

Objaśnianie szczegółów

- Wiersze dokumentujące pozwalają użytkownikom programów zorientować się czego dotyczy dana funkcja.
- Oprócz wierszy dokumentujących programista ma zawsze możliwość zamieszczenia w programie krótkich notatek, nazywanych komentarzami, które są przeznaczone dla niego bądź innych programistów i bardziej szczegółowo objaśniają pewne elementy programu.
  - Dowolny tekst umieszczony po znaku # aż do końca bieżącej linii jest traktowany przez Python jako komentarz.
  - Python nie stara się czytać, bądź rozumieć komentarza (chociaż super byłoby, gdyby to potrafił!).
- Precyzując, różnica pomiędzy łańcuchem dokumentującym a komentarzem jest taka, że łańcuch jest przeznaczony dla użytkownika danej funkcji. Użytkownik ten może nawet nie rozumieć programu.
  - Komentarz natomiast daje możliwość programiście podzielenia się znajomością szczegółów dotyczących działania programu i przekazania tej wiedzy innym ludziom, którzy, być może, będą chcieli zrozumieć lub zmodyfikować program.

Funkcja może być zapisana w więcej niż jednej linii

- Rozważona przez nas funkcja f miała tylko jedno polecenie (instrukcję, komendę), które podwajało argument funkcji i zwracało otrzymaną wartość.
- Jednak w ogólnym przypadku, funkcji w języku Python nie ogranicza się do jednego tylko polecenia. Mogą one składać z tylu poleceń ile aktualnie potrzeba.
  - Na przykład, mogło się zdarzyć że tę samą funkcję f napisano w taki sposób: :

```
def f(x):
    '''Takes a number x as an argument and returns 2*x.'''
    twoTimesX = 2 * x
    return twoTimesX
```

  - Zauważmy, że w sytuacji kiedy funkcja ma więcej niż jedno polecenie, to wszystkie te polecenia muszą być zapisane z jednakowym wcięciem. W ten sposób wskazujemy systemowi które polecenia wchodzą w skład funkcji, a które nie.

Funkcja może mieć więcej niż jeden argument

- Tak samo jak w matematyce, funkcje mogą mieć więcej niż jeden argument.
  - Oto przykładowa funkcja, która pobiera dwa argumenty x oraz y, a zwraca wartość $x^2 + y^2$:

```
def sumOfSquares (x , y):
    ''' Computes the sum of squares of its arguments '''
    return x**2 + y**2 # Here's our first comment!
```

- W istocie, argumenty funkcji nie muszą być liczbami.
  - Argumentem może być łańcuch znaków, lista i jeszcze parę innych rzeczy.
  - Oto kolejna funkcja, która jako argumenty pobiera dwa łańcuchy znakowe i zwraca także łańcuch.

```
def f_s(first, second):
        outputString = first + ' ' + second
        return outputString
```

```
>>> f_s('Hello', 'world')
'Hello world'
```

Użycie funkcji print

- W języku Python print jest funkcją wbudowaną, która pobiera argument (np. liczbę, łańcuch znaków lub dowolną inną wartość) i wyświetla go na ekranie.

```
>>> print(52.1)
52.1
```

```
>>> print("Hello world")
Hello world
```

```
>>> print([1, 2, 3])
[1, 2, 3]
```

```
>>> print(52,1)
52 1
```

- Funkcja print różni się od polecenia return w zasadniczy sposób.
  - Polecenie return nie jest funkcją – ono po prostu powoduje zakończenie działania funkcji i zwrócenie wyniku w miejsce, gdzie funkcja została wywołana.
  - Natomiast print jest funkcją, która wyświetla wskazaną wartość na ekran.
  - Wewnątrz pewnej funkcji można wielokrotnie użyć polecenia print. Każde z tych poleceń wyświetli to, co ma wyświetlić po czym funkcja wykonuje się dalej.
  - Z chwilą napotkania polecenia return funkcja zwraca wartość i kończy swoje działanie.

Po co pisać funkcje?

- W tym miejscu ktoś mógłby zapytać, z jakiego powodu pisze się funkcje.
  - Jeśli potrzebujemy podwoić liczby 10 and -1 to czy, zamiast pokonywać trudy definiowania funkcji, nie byłoby prościej wpisać operacje mnożenia jak w podanych tu przykładach?:

```
>>> 2 * 10
20
```

```
>>> 2 * -1
-2
```

  - W tym konkretnym przypadku, z pewnością bezpośrednie (lub nawet pamięciowe!) policzenie byłoby prostszym rozwiązaniem, jednak w przypadku ogólnym (jak się już wkrótce przekonamy), funkcje realizują o wiele bardziej złożone operacje, których każdorazowe wpisywanie byłoby prawdziwym utrapieniem.
  - Mechanizm funkcji daje możliwość wspólnego „spakowania” grupy obliczeń, co do których wiemy, że będziemy chcieli je wielokrotnie wykonywać.
  - Wspólne lokowanie pewnych obliczeń wewnątrz funkcji jest rodzajem abstrakcji: w ten sposób ukrywamy szczegóły (czyli polecenia tworzące tzw. ciało funkcji), co oznacza, że użytkownik wywołujący funkcję może skupić się na wyniku jej działania, a nie na tym, jak przebiegają obliczenia.

Instrukcje warunkowe – Problem Collatza

- Do tej pory nasze programy wykonywały proste obliczenia.
  - Niekiedy jednak pojawia się konieczność napisania programów, w których muszą być podejmowane decyzje, co oznacza, że wykonywane są różne operacje w zależności od spełnienia pewnego warunku.
- Rozważymy znany problem matematyczny nazywany problemem „3n+1" albo problemem Collatza.
  - Przedmiotem rozważań jest pewna funkcja, której argumentem jest liczba całkowita dodatnia n.
  - Jeśli wartością argumentu jest liczba parzysta, to funkcja zwraca wartość n/2. Jeśli natomiast argument jest liczbą nieparzystą, funkcja zwracaI 3n + 1.
  - Istotą problemu jest przypuszczenie, że niezależnie od jakiej liczby całkowitej dodatniej n rozpoczniemy, to wyznaczając dla kolejno otrzymywanych wyników wartość naszej funkcji, w końcu zawsze dojdziemy do liczby 1.
  - Na przykład rozpoczynając od n = 2, funkcja zwraca 1 bo n jest liczbą parzystą. Jeśli zaczniemy od n = 3, to pierwsze użycie funkcji zwróci 10. Dalej wyznaczając wartość funkcji dla 10 uzyskamy 5; dla 5 uzyskamy 16, co prowadzi dalej do 8, potem 4, potem 2 i na koniec 1!

Instrukcje warunkowe – Funkcja w języku Python (1/3)

- Sformułowana procedura może być zapisana w języku Python jako funkcja collatz(n):

```
def collatz( n ):
        '''Takes a single number as argument and applies Collatz to it'''
        if n % 2 == 0:
                return n/2
        else:
                return 3*n + 1
```

  - Zwróćmy uwagę na wyrażenie n % 2 == 0. Pamiętamy, że n % 2 wyznacza resztę z dzielenia n na 2. Jeśli n jest liczbą parzystą to wtedy ta reszta wynosi 0; jeśli nieparzystą, resztą jest 1.
  - Operator == wyznacza wartości wyrażeń po swojej lewej i prawej stronie i ustala, czy są one takie same. Cała konstrukcja z użyciem == (tutaj if n % 2 == 0) ma zawsze wartość True lub False.
  - Specjalne wartości True oraz False są nazywane wartościami logicznymi (boolowskimi), natomiast wyrażenie, którego wartością jest True lub False jest określane jako wyrażenie logiczne.

Instrukcje warunkowe – Funkcja w języku Python (2/3)

```
def collatz( n ):
        '''Takes a single number as argument and applies Collatz to it'''
        if n % 2 == 0:
                return n/2
        else:
                return 3*n + 1
```

☞ Pierwszym poleceniem funkcji collatz jest instrukcja warunkowa if.

- Bezpośrednio po słowie kluczowym if znajduje się wyrażenie boolowskie n % 2 == 0, a po nim następuje dwukropek. Python interpretuje tę instrukcję warunkową następująco: „Jeśli wyrażenie, które zostało mi dostarczone (w tym przypadku n % 2 == 0) ma wartość True, to wykonam wszelkie polecenia zapisane z „wcięciem" w kolejnych liniach". W naszym przypadku jest tylko jedna taka linia zawierająca polecenie zwrócenia wartości n/2.
- Z drugiej strony, jeśli testowane wyrażenie boolowskie ma wartość False, Python wykona wszelkie polecenia zapisane z „wcięciem" po linii else:. Można to potraktować jako opcję „w przeciwnym razie". Funkcja zwróci wtedy 3*n+1.

Instrukcje warunkowe – Funkcja w języku Python (3/3)

☞ Okazuje się, że konstrukcja else nie jest wymagana przez system i może być pominięta. Na przykład spójrzmy na taki oto, lekko zmodyfikowany, wariant funkcji collatz:

```
def collatz( n ):
  '''Takes a single number as argument and applies the Collatz to it'''
  if n % 2 == 0:
    return n/2
  return 3*n + 1
```

- Jeśli n jest parzyste, to funkcja wyznacza n/2 i zwraca tę wartość; zwrócenie wartości zatrzymuje działanie funkcji – co oznacza zakończenie wykonywania jakichkolwiek poleceń.
- To ostatnie zdanie jest ważne i często wprowadzające w zakłopotanie, dlatego warto podkreślić jeszcze raz: Wykonanie polecenia return zawsze powoduje natychmiastowe zatrzymanie działania funkcji. Zatem, jeśli n jest parzyste, Python nigdy nie dotrze do linii return 3*n + 1.
- Jeśli natomiast n jest liczbą nieparzystą to wyrażenie n % 2 ma wartość False. W takim przypadku Python „zjeżdża w dół" do pierwszej linii po poleceniu if, która ma taki sam poziom wcięcia, jak to polecenie if; w naszym przypadku jest to linia return 3*n+1. Widzimy zatem, że nowa wersja funkcji działa tak samo, jak wersja poprzednia z frazą (klauzulą) else.

Dopasowanie pierwszego znaku dwóch łańcuchów (1/3)

☞ Nasza następna funkcja będzie rozstrzygać, czy pierwszy znak w dwóch zadanych łańcuchach znaków jest taki sam, czy też nie. Na przykład:

```
>>> matchFirst ( 'spam' , 'super')
True
>>> matchFirst ( 'AAGC' , 'GAG')
False
```

- Oto jeden z możliwych sposobów napisania funkcji matchFirst:

```
def matchFirst ( s1 , s2 ):
  '''Compare the first characters in s1 and s2 and return True if they are the same. False if not'''
  if s1 [0] == s2 [0]:
    return True
  else :
    return False
```


Dopasowanie pierwszego znaku dwóch łańcuchów (2/3)

☞ Z reguły istnieje więcej niż jeden sposób napisania pewnego programu. Spójrzmy na krótszą (na pozór udziwnioną) wersję:

```
def matchFirst ( s1 , s2 ):
  '''Compare the first characters in s1 and s2 and return True if they are the same.
  False if not'''
  return s1[0] == s2[0]
```

- Co tutaj mamy? Warto pamiętać, że naszym celem jest wyznaczenie wartości boolowskiej — równejTrue lub False. Wyrażenie s1[0] == s2[0] ma wartość True lub False, i, niezależnie od tego, którą z nich w danej chwili przyjmuje, to jest właśnie to, co chcemy zwrócić. A zatem polecenie return s1[0] == s2[0] wytworzy pożądaną odpowiedź.

☞ Z drugiej jednak strony, w przypadku obu przedstawionych wersji funkcji matchFirst pojawia się delikatny problem: są pewne wartości argumentów, dla których te funkcje nie zadziałają.

- W szczególności, kiedy którykolwiek (bądź oba) z naszych argumentów okaże się pustym łańcuchem (to znaczy takim, który nie zawiera żadnych znaków) Python wyświetli komunikat o błędzie.

Dopasowanie pierwszego znaku dwóch łańcuchów (3/3)

- Możemy to naprawić dokładając jeszcze jedno polecenie if, które wykrywa ten szczególny przypadek, kiedy jeden lub oba argumenty są pustymi łańcuchami znaków.

```
def matchFirst ( s1 , s2 ):
  '''Compare the first characters in s1 and s2 and return True if they are the same.
  False if not'''
  if len(s1) == 0 or len(s2) == 0:
    return False
  else :
    return s1[0] == s2[0]
```

- Funkcja wbudowana len ustala, czy zadany łańcuch znaków ma długość 0. Łańcuch o długości 0 nie zawiera żadnych znaków, nawet tego z indeksem 0.
- Teraz, jeśli którykolwiek z łańcuchów jest pusty funkcja zwróci False.
- Zauważmy sposób użycia słowa or (*lub*) w poleceniu if. Polecenie to mówi: „jeśli długość s1 wynosi 0 lub długość s2 wynosi 0 to zwróć wartość False". Łącznik or to operator logiczny. Służy on do tworzenia wyrażeń logicznych, analogicznie jak dodawanie pozwala tworzyć wyrażenia arytmetyczne.

Inne operatory logiczne w języku Python

- Kolejnym przykładem operatora logicznego jest and. Naturalnym wydaje się to, że zwraca on True jeśli oba wyrażenia logiczne występujące po lewej i prawej stronie and są True oraz zwraca False w przeciwnym razie. Zatem polecenie:

```
len(s1) == 0 and len(s2) == 0
```

  będzie miało wartość True w przypadku, jeśli oba łańcuchy są puste.

- Wreszcie, w języku Python występuje się operator o nazwie not, który „neguje" wartość wyrażenia logicznego, zamieniając True na False, a False na True. Na przykład:

```
not 1 == 2
```

  ma wartość True, ponieważ 1 == 2 ma wartość False, co po zastosowaniu negacji daje True.

- Warto zaznaczyć, że możemy teraz zależnie od potrzeb stosować poznane operatory logiczne: or, and oraz not. Na przykład wyrażenie:

```
1 == 2 or not 41 == 42
```

  ma wartość True jako, że mimo iż 1 == 2 jest False, to not 41 == 42 jest True, a połączenie False oraz True operatorem or daje True.

Stosowanie wcięć w programie

- Zauważyliśmy już, że Python jest pedantyczny w kwestii stosowania wcięć przy pisaniu kodu.
  - System wymaga, aby po pierwszej linii kodu (tej, która rozpoczyna się słowem def), cała reszta funkcji była zapisana z użyciem wcięcia (tabulacji).
- Wcięcia stosuje się także w poleceniach if oraz else.
  - Z wcięciem zapisujemy linię lub linie, które znajdują się bezpośrednio po poleceniu if i mają być wykonane w przypadku kiedy wyrażenie logiczne ma wartość True. W podobny sposób stosujemy wcięcie dla linii, które mają być wykonane w sekcji else.
- Kod pokazany obok przedstawia możliwy alternatywny sposób napisania naszej funkcji collatz; w tym wariancie w jednej linii wyznacza się wymaganą wartość, która następnie, w drugiej linii, jest zwracana.

```
def collatz(n):
    '''Takes a single number as an argument and
       applies the Collatz function to it.'''
    if n % 2 == 0:
        result = n/2     # Create a variable called result
        return result    # Now we return the value of result
    else:
        result = 3*n + 1    # Create a variable called result
        return result       # Now return the value of result
```


Odległość edycyjna dwóch łańcuchów znakowych

- Rozważymy teraz dwa łańcuchy znaków “spam” and “poems”. Możliwym sposobem przejścia od “spam” do “poems” jest wykonanie następującego ciągu ciągu operacji:
  - usuń “s” ze “spam” aby otrzymać “pam” (usuwanie),
  - zamień “a” na “o” aby otrzymać “pom” (zamiana),
  - wstaw “e” po “o” aby otrzymać “poem” (wstawianie), oraz
  - wstaw “s” na końcu “poem” aby otrzymać “poems” (jeszcze jedno wstawianie).
- Potrzebne było wykonanie czterech operacji.
  - Rzeczywiście, jest to najmniejsza liczba operacji wymagana przy przejściu od “spam” do “poems.” Inaczej mówiąc, odległość edycyjna pomiędzy “spam” i “poems” jest równa 4.
  - Zauważmy, że właśnie zdefiniowaliśmy odległość edycyjną jako najmniejszą liczbę operacji potrzebnych do przejścia od pierwszego do drugiego łańcucha.
  - Po chwili zastanowienia przekonujemy się, że dokładnie tyle samo operacji wymagałoby przejście od drugiego łańcucha do pierwszego. Inaczej mówiąc, odległość edycyjna jest symetryczna – tzn. taka sama, niezależnie od którego spośród dwóch łańcuchów wyjdziemy.

Wielokrotne użycie warunku (1/2)

- W niektórych przypadkach przydaje się użycie więcej niż jednej alternatywy do warunku testowanego w poleceniu if.
- Rozważymy zadanie wyznaczania odległości edycyjnej. Chociaż nie jesteśmy jeszcze całkiem gotowi aby rozwiązać go w pełni, możemy ograniczyć się do przypadku, kiedy jeden lub oba łańcuchy są puste.
  - W takim przypadku odległość edycyjna łańcuchów jest po prostu równa długości tego który nie jest pusty, dlatego, że taka właśnie liczba operacji wstawiania do pustego łańcucha (albo usuwania z niepustego) będzie konieczna do tego aby otrzymać dwa takie same łańcuchy.

- Oto funkcja wyznaczająca odległość edycyjną dla rozważanego tu prostego przypadku:

```
def simpleDistance(s1, s2):
    '''Takes two strings as arguments and returns the edit
       distance between them if one of them is empty.
       Otherwise it returns an error string.'''
    if len(s1) == 0:
        return len(s2)
    elif len(s2) == 0:
        return len(s1)
    else:
        return 'Help! We don''t know what to do here!'
```


Wielokrotne użycie warunku (2/2)

```
def simpleDistance(s1, s2):
    '''Takes two strings as arguments and returns the edit
       distance between them if one of them is empty.
       Otherwise it returns an error string.'''
    if len(s1) == 0:
        return len(s2)
    elif len(s2) == 0:
        return len(s1)
    else:
        return 'Help! We don''t know what to do here!'
```

- Funkcja simpleDistance rozpoczyna działanie od sprawdzenia, czy łańcuch s1 jest pusty; jeśli tak, to jest zwracana długość s2. Jeśli s1 nie jest pusty, to funkcja użyje polecenia elif, aby sprawdzić czy s2 jest pusty. Jeśli tak, to zostanie zwrócona długość s1. Ostatecznie, jeśli żaden spośród łańcuchów s1 i s2 nie jest pusty, zostanie zwrócony komunikat, że „nie wiadomo co robić”.

- Zauważmy, że jeśli oba łańcuchy są puste, to zostanie zwrócone 0, co jest poprawną odpowiedzią.
- Polecenia elif należy wymawiać jako “else if”. Sekcja elif w tej funkcji zadziała tylko wtedy jeśli łańcuch s1 nie był pusty. Polecenie ze słowem elif oznacza tyle co “inaczej (w przeciwnym razie), jeśli łańcuch s2 jest pusty, to zwróć długość łańcucha s1.”
- Chociaż w tej akurat funkcji konstrukcji elif użyto tylko raz, to, w przypadku ogólnym, po if możemy użyć warunku elif tyle razy, ile potrzeba. Potem, na końcu, może pojawić się zero lub jedno polecenie else, ale nie więcej niż jedno! Końcowe polecenie else ustala co robić, jeśli wszystkie warunki nie są spełnione.

Przypadek dwóch łańcuchów o długości 4

- A jak będzie w przypadku, kiedy żaden z łańcuchów nie jest pusty?
- Załóżmy póki co, że oba łańcuchy mają dokładnie po cztery znaki.
  - W takim przypadku do przejścia od 1-go do 2-go łańcucha nie będą potrzebne żadne operacje usuwania i dodawania znaków – wystarczą zamiany. Na przykład, aby przejść od „spam” do „spim” wystarczy zamienić „a” na „i”.
  - Odległość edycyjną dwóch łańcuchów 4-znakowych oblicza następująca funkcja:

```
def distance(s1, s2):
    '''Return the distance between two strings,
       each of which is of length four.'''
    editDist = 0
    if s1[0] != s2[0]:
        editDist = editDist + 1
    if s1[1] != s2[1]:
        editDist = editDist + 1
    if s1[2] != s2[2]:
        editDist = editDist + 1
    if s1[3] != s2[3]:
        editDist = editDist + 1
    return editDist
```

- Zapis `!=`, symbolizujący relację „nie równa się” , jest o wiele bardziej zgrabną notacją od znaczącego to samo `not s1[0] == s2[0]`. Zauważmy, że funkcja rozpoczyna od nadania zmiennej o nazwie `editDist` wartości `0`, a następnie dodaje `1` do tej zmiennej każdorazowo, kiedy napotyka na parę odpowiadających sobie znaków, które nie są identyczne i wymagają jednej operacji zamiany.
- Chociaż program ten będzie działał dla 4-znakowych słów, to nie poradzi sobie w przypadku słów dłuższych lub krótszych. Ponadto nie jest w stanie obsłużyć sytuacji dodawania i usuwania znaków.

Rekurencja (1/3)

- Aby możliwa była obsługa zarówno łańcuchów dowolnej długości, jak i operacji wstawiania i usuwania znaków, potrzebne będzie użycie nowej jakości jaką stanowi rekurencja (rekursja).
- Zanim użyjemy rekursji do wyznaczania odległości edycyjnej, wyobraźmy sobie grupę złożoną z, na przykład, 42 studentów, których należy ustawić w jednym szeregu. Na ile sposobów można to zrobić?
  - Być może znamy już odpowiedź: liczba ta wynosi $42\times41\times40\ldots3\times2\times1$ i jest określana terminem „42 silnia” oraz zapisywana jako $42!$. (Istnieje 42 możliwości wyboru studenta, który ma zająć pierwszą pozycję w szeregu, 41 możliwości wybrania kogoś na następne miejsce i tak dalej).
- A jak napisać program w języku Python, który oblicza silnię dla dowolnej liczby całkowitej dodatniej?
  - Na szczęście zauważamy, że $42!=42\times41!$ i, ogólnie, $n!=n(n-1)!$. Oznacza to, że silnia liczby n może być otrzymana z mnożenia n razy wynik rozwiązania innego, mniejszego zadania obliczania silni. Tak sformułowany przepis na obliczanie silni nazywamy definicją rekurencyjną: wyraża ona dany problem używając przy tym mniejszej wersji tego samego problemu; określenie pojawia się ponownie (recur) w rozwiązaniu.

Rekurencja (2/3)

- Zanim przedstawimy program obliczający silnię, zauważmy póki co, że definicja rekurencyjna rzeczywiście ma sens praktyczny.
  - Przypuśćmy, że chcemy obliczyć 3!. Zgodnie z definicją rekurencyjną wielkość ta wynosi 3×2!. A zatem teraz „udajemy się", aby obliczyć 2!. Używając ponownie tej samej definicji widzimy, że 2!=2×1!. Dalej będziemy zachwyceni jeśli uda się wyznaczyć 1!. Zgodnie z definicją, 1! jest równe 1×0!. Nadal posługując się definicją, 0! to 0×(−1)!... Niestety, nie jest dobrze—wygląda na to, że ten proces nie kończy się... Ponadto, nie interesuje nas 0!, ani silnia liczby ujemnej.
- Aby obejść trudność, jaka się pojawiła, należy dodać regułę, która określi, kiedy należy zakończyć proces rekurencyjny.
  - Sensownym wydaje się następujące zalecenie: „Jeśli dojdziesz do miejsca, w którym trzeba obliczyć 1!, zakończ stosowanie rekurencji i po prostu zwróć wynik w tym przypadku wynoszący 1." Jest to przypadek bazowy definicji rekurencyjnej. Informuje on, kiedy zakończyć stosowanie reguły rekurencyjnej.
  - Rzecz jasna, należy ciągle sprawdzać przypadek bazowy, zanim zostanie podjęta decyzja o kontynuacji stosowania reguły rekurencyjnej.

Rekurencja (3/3)

- Wracając do przykładu obliczania 3!,
  - Schodzimy w dół do poziomu 1! a przypadek bazowy oznajmia „Stop! To jest 1." A zatem, mamy 1!=1.
  - Pamiętajmy że wartość 1! była potrzebna w miejscu obliczania 2!=2×1!. A zatem teraz wstawiamy 1 za 1! uzyskując 2!=2×1=2.
  - Z kolei obliczenie 3!=3×2! prosiło o 2! i cierpliwie czeka na odpowiedź. Kiedy ta odpowiedź pojawia się, jesteśmy w stanie ustalić, że wartością 3!jest 3×2=6.
  - A zatem wyznaczyliśmy wartość 3! wykorzystując definicję rekurencyjną.
- Spróbujmy teraz odtworzyć definicję rekurencyjną w języku Python na tyle wiernie, na ile to możliwe.
  - Program może na pierwszy rzut oka wydać się dziwnym lub nawet całkowicie błędnym, ale wypróbujmy go.

```
def factorial(n):
    '''Recursive function for computing
    the factorial of n.'''
    if n == 1:
        return 1
    else:
        result = n * factorial(n-1)
        return result
```

Funkcja rekurencyjna

- Typowa funkcja rekurencyjna składa się z dwóch podstawowych elementów:
  - Przypadek bazowy : Określa wartość zwracaną przez tę funkcję dla „najprostszego" argumentu.
  - Krok rekurencji : Oznacza rozwiązanie mniejszej wersji zadania dzięki wywołaniu przez funkcję „samej siebie" ale z mniejszym (prostszym) argumentem. To rozwiązanie mniejszego problemu zostaje następnie w pewien sposób wykorzystane do rozwiązania problemu początkowego.
- Dla naszej funkcji obliczania silni
  - przypadkiem bazowym jest sytuacja kiedy `n` jest równe `1`. Wówczas funkcja po prostu zwraca `1`, ponieważ `1!=1`. Na tym funkcja kończy swoje działanie.
  - Krok rekurencji, reprezentowany przez fragment umieszczony wewnątrz polecenia `else`, oblicza silnię liczby `n-1`, mnoży uzyskany wynik przez `n`, a następnie zwraca otrzymaną wartość.

Sekret związany z możliwością użycia rekurencji polega na postawieniu sobie pytania: „Czy pomocnym okaże się uzyskanie rozwiązania tego samego problemu ale w mniejszej wersji?" Jeśli odpowiedzią jest „tak", to możemy napisać funkcję rekurencyjną, która wywołuje samą siebie celem rozwiązania mniejszej wersji zadania, a następnie wykorzystuje otrzymany wynik do rozwiązania problemu początkowego.

Odległość edycyjna dla łańcuchów jednakowej długości (1/2)

- Rozważmy ponownie funkcję obliczającą odległość edycyjną. Ciągle nie jesteśmy gotowi aby problem rozwiązać w pełni (dla dowolnych łańcuchów), ale przybliżamy się do tego!
- Zajmiemy się teraz następującym, bardziej ogólnym, przypadkiem:
  - Oba łańcuchy mają zawsze tę samą długość (a zatem nie będą wymagane żadne operacje wstawiania ani usuwania)
  - Tym razem jednak ich długość może być dowolna – nie tylko równa 4! W takim przypadku odległość edycyjną stanowi liczba pozycji na których oba łańcuchy różnią się od siebie.
- Przypadkiem bazowym jest teraz sytuacja, kiedy oba łańcuchy są puste – jest to najprostszy możliwy przypadek, kiedy mają one tę samą długość.
  - Wtedy odległość edycyjna wynosi `0`. Funkcja kończy się wykonywać i zwraca `0` w miejsce, skąd została wywołana.

Odległość edycyjna dla łańcuchów jednakowej długości (2/2)

- Jeśli przypadek bazowy nie znajduje zastosowania – to znaczy łańcuchy mają pewną niezerową długość – sposób dalszego postępowania zależy od tego, czy znak na pozycji 0 jest w tych łańcuchach taki sam.
  - Jeśli nie, ta pozycja „dokłada" 1 do odległości edycyjnej.
  - Teraz musimy porównać ze sobą pozostałą część obu łańcuchów, aby wyznaczyć liczbę miejsc na których się one różnią. Ale przecież jest to dokładnie ten sam problem, tyle tylko, że dla dwóch łańcuchów, w których „obcięto" pierwsze znaki!
  - A zatem jeśli znaki na pozycji 0 nie są takie same, odległość edycyjna pomiędzy s1 and s2 wynosi 1 plus odległość edycyjna pomiędzy s1[1:] i s2[1:]. Przypomnijmy, że zapis s1[1:] oznacza łańcuch s1 ale bez znaku na pozycji 0.
  - Z drugiej strony, w sytuacji gdy pierwszy znak w obu łańcuchach jest taki sam, odległość edycyjna pomiędzy s1 i s2 jest po prostu odległością edycyjną pomiędzy s1[1:] i s2[1:].
  - Funkcję w języku Python realizującą taki przepis pokazano obok.

```
def simpleDistance(s1, s2):
    '''Takes two strings of the same length and returns the
       number of positions in which they differ.'''
    if len(s1) == 0:    # len(s2) is also 0 since strings
                        # have the same length
        return 0        # base case
    elif s1[0] != s2[0]: # recursive step, case 1
        return 1 + simpleDistance(s1[1:], s2[1:])
    else:               # recursive step, case 2:
                        #   s1[0] == s2[0]
        return simpleDistance(s1[1:], s2[1:])
```


Odwracanie łańcucha (1/2)

- Aby odwrócić kolejność znaków w dowolnym łańcuchu, można odciąć pierwszy znak, odwrócić resztę łańcucha, a następnie dodać odcięty pierwszy znak na koniec.
- Stosują tę regułę widzimy, że odwracanie łańcucha "spam" wymaga wykonania następujących operacji
  - Odwrócimy najpierw "pam" (a później dodamy "s" na koniec).
  - Aby odwrócić "pam", odwracamy najpierw "am" (a później dodajemy "p" na koniec).
  - Aby odwrócić "am", odwracamy najpierw "m" (a później dodajemy "a" na koniec).
  - A co robimy aby odwrócić "m"? No cóż, kontynuując w tym duchu, obcinamy "m" i zastajemy z ""—łańcuchem nie zawierającym znaków, określanym jako łańcuch pusty.
- Jak odwrócimy łańcuch pusty?
  - Bardzo prosto: to jest przypadek bazowy naszej rekurencji. Jeśli zostaniemy poproszeni o odwrócenie łańcucha pustego, to wynikiem będzie w sposób ewidentny ten właśnie łańcuch pusty!

Odwracanie łańcucha (2/2)

☞ Rozważmy następującą funkcję rekurencyjną w języku Python, która pobiera jako argument dowolny łańcuch znaków i zwraca w wyniku odwróconą wersję tego łańcucha:

```
def reverse(string):
  '''Takes a string as an argument and returns
    its reversal.'''
  if string == '':          # Is the string empty?
    return ''               # If so, reversing it is easy!
  else:
    firstSymbol = string[0] # Hold on to the first symbol
    return reverse(string[1:]) + firstSymbol
```

☞ Jeśli łańcuch jest pusty, zadanie jest łatwe – po prostu zwracamy pusty łańcuch i gotowe.

- Jest to przypadek bazowy i zajmujemy się nim w pierwszej kolejności.

☞ Jeśli łańcuch nie jest pusty, odnajdujemy pierwszy znak i zachowujemy go w „bezpiecznym miejscu" do późniejszego wykorzystania.

- Innymi słowy użyjemy zmiennej, która przechowa ten pierwszy znak do czasu, kiedy będzie potrzebny.
- Następnie obcinamy ten pierwszy znak, odwracamy pozostałą część łańcucha, i dodajemy zachowany pierwszy symbol na koniec łańcucha wynikowego.

Metoda „użyj albo odrzuć" (1/7)

☞ Jesteśmy prawie gotowi do pełnego rozwiązania problemu odległości edycyjnej.

- W istocie, dysponujemy wszelkimi potrzebnymi do tego celu narzędziami programowania.
- Jednakże, problem odległości edycyjnej dla ogólnego przypadku, jest o wiele bardziej złożony niż odwracanie łańcucha i brakuje nam pewnych narzędzi do rozwiązywania problemów, które w istotny sposób ułatwiłyby sprawę.

☞ Stąd też zaprezentujemy teraz pewne ogólne podejście do rozwiązywania obszernej klasy zadań (w tym także problemu odległości edycyjnej), nazywane metodą „użyj albo odrzuć".

☞ Przypuśćmy, że chcemy wybrać zestaw (podzbiór) przedmiotów, których całkowita waga jest możliwie maksymalnie bliska wytrzymałości walizki lecz jej nie przekracza.

- Wyobraźmy sobie, na przykład, że wytrzymałość walizki jest równa 42 jednostkom i mamy przedmioty ważące 5, 10, 18, 23, 30 i 45 jednostek. W tym przypadku najlepsze co możemy zrobić jest wybranie przedmiotów ważących 18 i 23, co ogółem daje 41. Z drugiej strony, jeśli mielibyśmy także przedmiot o wadze 2, moglibyśmy otrzymać dokładnie 42, wybierając przedmioty o wadze 2, 10 oraz 30.

Metoda „użyj albo odrzuć” (2/7)

- A zatem, to czego potrzebujemy, to funkcja, która
  - pobiera dwa argumenty: liczbę reprezentującą wytrzymałość (nośność) walizki oraz listę liczb dodatnich (bez konkretnego uporządkowania) reprezentujących wagi przedmiotów.
  - Funkcja ta powinna zwracać największą wagę całkowitą przedmiotów, które mogą być wybrane bez przekroczenia wytrzymałości walizki.
- Funkcję nazwiemy subset a jej użycie (kiedy już powstanie!) wyobrażamy sobie następująco:

```
>>> subset(42, [5, 10, 18, 23, 30, 45])
41
>>> subset(42, [2, 5, 10, 18, 23, 30, 45])
42
```

  - Nieco bardziej ambitnym zadaniem byłoby dostarczenie zbioru przedmiotów, które dają najlepsze rozwiązanie, ale, póki co, ten problem zostawiamy.

Metoda „użyj albo odrzuć” (3/7)

- Najpierw zajmiemy się przypadkiem bazowym.
  - Czym są te „łatwe” przypadki, kiedy funkcja subset prawie nie musi nic robić?
  - Dosyć łatwo zauważyć, że jeśli zadana wytrzymałość jest równa 0, nie możemy wybrać żadnego z przedmiotów, a zatem należy zwrócić 0 co będzie oznaczać „przepraszam, ale maksymalną sumą którą możesz zebrać jest 0”.
  - A zatem, możemy rozpocząć następująco:

```
def subset(capacity, items):
    '''Given a suitcase capacity and a list of items
       consisting of positive numbers, returns a number
       indicating the largest sum that can be made from a
       subset of the items without exceeding the capacity.'''

    if capacity == 0:
        return 0
```

  - Właściwie możemy także zwrócić zero jeśli wytrzymałość jest mniejsza od zera, co zobaczymy dalej.

Metoda „użyj albo odrzuć” (4/7)

- Istnieje wszakże jeszcze jeden „łatwy” przypadek dotychczas przez nas nie rozpatrywany. Co się dzieje w przypadku, gdy lista przedmiotów jest pusta? Wtedy także nie możemy niczego stamtąd zabrać. Można to uwzględnić umieszczając elif po poleceniu if:

```
elif items == []:
    return 0
```

- Zamiast tego, skoro zamierzamy zwrócić 0 w przypadku gdy wytrzymałość jest mniejsza lub równa 0 lub lista jest pusta, możemy po prostu zmodyfikować poprzednie polecenie if następująco:

```
if capacity <= 0 or items == []:
    return 0
```

☞ Pamiętamy sugestię aby w pierwszej kolejności zająć się przypadkami bazowymi, gdyż dotyczą one „łatwych” sytuacji.

- Zauważmy, że skoro funkcja subset ma dwa argumenty, to należy spodziewać się dwóch przypadków bazowych: każdy z argumentów może pojawić się w łatwym wariancie (capacity jest ≤0 lub lista items jest pusta).
- Często liczba przypadków bazowych jest równa liczbie argumentów funkcji.

Metoda „użyj albo odrzuć” (5/7)

☞ Zajmijmy się pierwszym elementem w zadanej liście.

- W języku Python jest to bardzo proste gdyż tym elementem jest items[0].

☞ Gdyby zdarzyło się, że waga pierwszego przedmiotu przewyższa wytrzymałość walizki, to wtedy nie pozostaje nic innego jak wyłączyć (odrzucić) ten przedmiot.

- W takim przypadku stajemy przed prostszym zadaniem: Znaleźć najlepsze rozwiązanie dla zadanej wytrzymałości, ale rozważyć listę, w której usunięto pierwszy element.
- Chcielibyśmy w takim przypadku wywołać pewną funkcję, która pomogłaby znaleźć takie rozwiązanie. Jakiej funkcji moglibyśmy użyć? Otóż jest nią nasza funkcja subset! Pokazany obok kod odzwierciedla dotychczasowe ustalenia:

```
def subset(capacity, items):
    '''Given a suitcase capacity and a list of items
       consisting of positive numbers, returns a number
       indicating the largest sum that can be made from a
       subset of the items without exceeding the capacity.'''

    if capacity <= 0 or items == []:
        return 0
    elif items[0] > capacity:
        return subset(capacity, items[1:])
```

Metoda „użyj albo odrzuć” (6/7)

- Na koniec, jeśli rozważany pierwszy przedmiot items[0] nie przewyższa aktualnej wytrzymałości, możemy zdecydować się na jego użycie bądź też nie.
  - Na przykład jeśli wytrzymałość wynosi 10 a listę przedmiotów stanowi [8, 4, 6] moglibyśmy użyć przedmiotu o wartości 8, ale jeśli to zrobimy, to nie możemy wtedy użyć żadnego z pozostałych (dlatego, że każdy z pozostałych przedmiotów w liście przewyższa pozostałą wytrzymałość). Lepszym rozwiązaniem byłoby nie użyć tego przedmiotu i wziąć przedmioty o wartościach 4 i 6 uzyskując wynik równy 10.
- A zatem pytanie stawiane tutaj brzmi: „użyć czy odrzucić” (w przykładzie dotyczyło ono przedmiotu o wadze 8).
  - Na razie nie znamy odpowiedzi na to pytanie.
  - Niemniej możemy użyć dwóch wywołań rekurencyjnych, z których jedno znajduje najlepsze rozwiązanie w przypadku „użycia” przedmiotu o wartości 8, a drugie znajduje najlepsze rozwiązanie w przypadku „odrzucenia” przedmiotu o wartości 8.
  - Lepsze z tych dwóch rozwiązań musi być rozwiązaniem najlepszym.
  - Tak czy inaczej, jeśli chodzi o nasz przedmiot o wartości 8, to dowolne rozwiązanie musi go albo użyć albo odrzucić!

Metoda „użyj albo odrzuć” (7/7)

- Rekurencja stara się „ocenić” obie te możliwości.
  - Przypadek, kiedy odrzucamy pierwszy przedmiot z listy jest stosunkowo prosty. Wtedy należy po prostu znaleźć najlepsze rozwiązanie dla takiej samej wytrzymałości oraz listy items[1:].
  - Jeśli zdecydujemy się na użycie tego przedmiotu, to wówczas jego waga pojawia się w naszym rozwiązaniu, ale wytrzymałość zostaje pomniejszona o wartość wagi.
  - A zatem gotowa funkcja ma następującą postać:

```
def subset(capacity, items):
    if capacity <= 0 or items == []:
        return 0
    elif items[0] > capacity:
        return subset(capacity, items[1:])
    else:
        loseIt = subset(capacity, items[1:])
        useIt = items[0] + subset(capacity - items[0], items[1:])
        return max(loseIt, useIt)
```

- Użyta w kodzie funkcja max jest funkcją wbudowaną języka Python; pobiera ona dowolną liczbę argumentów oraz zwraca ich maksymalną wartość.

Rozwiązywanie problemów metodą „użyj albo odrzuć”

- Przedstawiona koncepcja „użyj albo odrzuć” jest skuteczną strategią rozwiązywania problemów.
  - Umożliwia ona zastosowanie rekurencji do zbadania wszelkich możliwych rozwiązań.
- Po tym jak poznaliśmy rekurencję oraz strategię „użyj albo odrzuć”, jesteśmy gotowi aby rozwiązać postawiony wcześniej problem znalezienia odległości edycyjnej dwóch łańcuchów.
  - Celem szybkiego przypomnienia, prześledzimy przykład, który pokazuje ten problem w pełnej wersji. Załóżmy, że należy przekształcić łańcuch „alien” w łańcuch „sales”. Możemy rozpocząć wstawiając „s” na początek „alien” co prowadzi do „salien”. Następnie usuwamy „i” uzyskując „salen”. Dalej zastępujemy „n” na „s” aby otrzymać „sales”. Potrzebne były trzy operacje i rzeczywiście nie jest możliwe przejście od „alien” do „sales” używając mniej niż trzech operacji.
  - Pamiętajmy, że celem jest znalezienie najmniejszej liczby operacji zamiany, wstawiania oraz usuwania wymaganych do przejścia od jednego łańcucha do drugiego.
  - Ponieważ nie ma znaczenia czy decydujemy się zamienić pierwszy łańcuch na drugi czy odwrotnie, przyjmijmy, że wychodzimy od pierwszego łańcucha, który staramy się przekształcić na łańcuch drugi.

Wyznaczenie odległości edycyjnej dwóch łańcuchów (1/4)

- Naszym celem jest teraz napisanie funkcji o nazwie distance która pobiera dwa łańcuchy jako argumenty; niech będą to łańcuchy first oraz second.
- W proponowanym rozwiązaniu zostanie użyta rekurencja, dlatego zaczniemy od przypadku bazowego. Ponieważ mamy dwa argumenty, należy spodziewać się dwóch przypadków bazowych, odnoszących się do sytuacji, które możemy określić jako „łatwe” lub „graniczne”.
  - Zdarzeniem granicznym jest sytuacja, kiedy jeden (lub oba) łańcuchy są puste. Załóżmy, na przykład, że pierwszy z nich jest pusty. Na razie przyjmujemy, że drugi nie jest pusty – może to być na przykład „spam”. Wtedy odległość pomiędzy tymi dwoma łańcuchami musi być równa długości drugiego łańcucha, gdyż taką właśnie ilość znaków należy wstawić do pustego łańcucha, aby otrzymać ten drugi. Podobnie, jeśli to drugi łańcuch jest pusty, wówczas odległość musi być równa długości pierwszego, ponieważ taką właśnie liczbę znaków musimy usunąć. A zatem zapiszmy początkową wersję naszej funkcji tak:

```
def distance(first, second):
    '''Returns the edit distance between first
    and second.'''

    if first == '':
        return len(second)
    elif second == '':
        return len(first)
```

  - To jeszcze nie wszystko, ale warto byłoby teraz zapytać, co się stanie w przypadku, kiedy oba łańcuchy są puste. Wtedy odległość powinna wynosić 0; zauważmy że taki właśnie wynik otrzymamy, ponieważ pierwszy łańcuch jest pusty to dostajemy długość drugiego – która jest równa 0. Zawsze warto przyjrzeć się takim szczególnym sytuacjom, aby upewnić się, że czegoś nie przeoczyliśmy.

Wyznaczenie odległości edycyjnej dwóch łańcuchów (2/4)

- Wróćmy do rekurencji.
- Jeśli łańcuchy first i second rozpoczynają się od tego samego znaku, to możemy uznać, że mamy szczęście i nie zajmować się dalej tymi jednakowymi znakami.
  - W tym przypadku odległość pomiędzy dwoma rozważanymi łańcuchami jest po prostu odległością pomiędzy tymi samymi łańcuchami, ale z obciętym pierwszym znakiem. Oznacza to, że musimy wyznaczyć odległość pomiędzy first[1:] a second[1:].
  - Na przykład, wyznaczając odległość pomiędzy „spam" i spim" (zauważmy , że wynosi ona 1), fakt, że oba rozważane łańcuchy rozpoczynają się od tej samej litery pozwala wnioskować, że poszukiwana odległość jest taka sama jak odległość pomiędzy „pam" i „pim". Prowadzi to do następującego wywołania rekurencyjnego: distance (first[1:], second[1:]).
- Jeśli łańcuchy rozpoczynają się od różnych znaków, to problem okazuje się bardziej interesujący.
  - Skoro pierwsze znaki różnią się konieczna będzie korekta. Możemy w szczególności zamienić pierwszy znak pierwszego łańcucha tak, aby był on zgodny z pierwszym znakiem drugiego łańcucha (zamiana), usunąć pierwszy znak pierwszego łańcucha (usuwanie), bądź też dodać nowy znak na początek pierwszego łańcucha (wstawianie). Nie wiemy, która z tych operacji jest najbardziej korzystna, dlatego użyjemy rekursji do zbadania wszystkich trzech opcji. Sytuacja nawiązuje do strategii „użyj albo odrzuć" z tym, że teraz mamy do wyboru trzy możliwości a nie dwie.

Wyznaczenie odległości edycyjnej dwóch łańcuchów (3/4)

- Przedyskutujmy kolejno każdą z trzech opcji.
- Jeśli zdecydujemy się na zamianę, wtedy nietrudno zauważyć, że należy zamienić pierwszy znak w łańcuchu first tak, aby był on taki sam jak pierwszy znak w łańcuchu second.
  - Teraz pierwsze znaki będą zgodne i możemy więcej się nimi nie zajmować. Dlatego najlepsze rozwiązanie rozpoczynające się od zamiany będzie miało koszt wynoszący 1 + distance(first[1:], second[1:]), gdyż liczba operacji będzie sumą jedynki (zamiana) plus pewna liczba operacji potrzebnych do przejścia od łańcucha first[1:] do łańcucha second[1:].
- Drugą opcją jest usunięcie pierwszego znaku łańcucha first.
  - Potrzeba na to jednej operacji, po której pozostaje jeszcze wyznaczyć odległość pomiędzy first[1:] a second.
- Na koniec należy rozważyć wstawienie nowego znaku na początek łańcucha first.
  - Nietrudno zauważyć, że ten nowy znak powinien być zgodny z pierwszym znakiem w second. Będzie to wymagało jednej operacji, po której pozostaje wyznaczenie odległości od first do second[1:]. Takie wyliczenie wynika stąd, że nie użyliśmy jeszcze pierwszego znaku w first a pierwszy znak w second został właśnie dopasowany.

Wyznaczenie odległości edycyjnej dwóch łańcuchów (4/4)

- Najlepsza spośród trzech omówionych opcji będzie naszym najlepszym rozwiązaniem!
  - Po zebraniu dotychczasowych ustaleń w jedną całość otrzymujemy następującą funkcję:

```
def distance(first, second):
    '''Returns the edit distance between first and second.'''

    if first == '':
        return len(second)
    elif second == '':
        return len(first)
    elif first[0] == second[0]:
        return distance(first[1:], second[1:])
    else:
        substitution = 1 + distance(first[1:], second[1:])
        deletion = 1 + distance(first[1:], second)
        insertion = 1 + distance(first, second[1:])
        return min(substitution, deletion, insertion)
```

- Podobnie jak max, min jest funkcją wbudowaną języka Python, która zwraca minimalną wartość wyznaczoną spośród jej argumentów.
- Oto kilka przykładów użycia funkcji distance:

```
>>> distance('','')
0
>>> distance('','klasa')
5
>>> distance('123','')
3
>>> distance('rysunek','rabunek')
2
>>> distance('profesor','proces')
3
```


Programowanie funkcyjne

- Omówione przykłady, poprowadziły nas od absolutnych podstaw języka Python do pisania efektywnych funkcji rekurencyjnych.
- Taki styl programowania – używanie funkcji i rekurencji – jest nazywany programowaniem funkcyjnym.
- Co ciekawe, zdobyte do tej pory umiejętności wystarczają do napisania dowolnego programu.
  - Dokładniej mówiąc, poznane cechy Pyhona są wystarczające do napisania każdego programu, który może być napisany w jakimkolwiek innym języku.
  - Może wydać się dziwnym dlaczego zaryzykowano tak stanowcze stwierdzenie. W odpowiedzi można powołać się na pasjonujący dział informatyki nazywany teorią obliczeń, dzięki której można rzeczywiście udowodnić tego typu deklaracje.
- Wprawdzie potrafimy już programować, ale istnieją jeszcze inne interesujące pomysły umożliwiające pisanie bardziej efektywnych, zwięzłych i eleganckich programów.
  - Wcześniej jednak warto poćwiczyć programowanie funkcjonalne i użycie rekurencji, pisząc własne programy.