Wstęp do programowania
semestr zimowy 2024/2025

Dr Anna Muranova
UWM w Olsztynie

Wykład 7

Czytanie z plików

Proces czytania i zapisu danych do pliku to zadanie złożone. Python jak większość języków programowania pozwala wczytywać dane ze zbiorów zewnętrznych, jak i zapisywać dane do plików. Proces czytania można zrealizowaćć przy pomocy funkcji wbudowanych -`open()`, `read()` i `close()` lub – w przypadku danych o ustalonej strukturze – najczęściej tabelarycznej, można skorzystać z gotowych funkcji wspierających ten proces.

Aby odczytać plik tekstowy należy wykonać trzy polecenia:

- `open()` aby nawiązać połączenie z plikiem (nic nie zostaje wczytane)
- `read()` aby wczytać całą zawartość pliku do jednej zmiennej jako tekst
- `close()` zamknąć plik po zakończeniu czytania

Dodatkowo proces czytania lub zapisu wymaga znalezienia właściwego pliku, lub - w przypadku odczytu utworzenia nowego pliku.

Dalej używa się plik.txt z zawartością

abrakadabra
abra
kadabra

Czytanie z plików

Wczytanie niewielkiego pliku tekstowego znajdującego się w tym samym katalogu co nasz skrypt

```
f = open("plik.txt")
zawartosc = f.read()
f.close()
print(zawartosc)
```

```
abrakadabra
abra
kadabra
```

Funkcja `open()` nawiązuje połączenie z plikiem, ale nie wczytuje danych. Przypisuje jedynie do obiektu f wskaźnik do pliku. Wynika to z faktu że zbiór danych może być bardzo duży i programista powinien zachować kontrolę nad jego wczytywaniem. Funkcja `read()` wczytuje całą zawartość ze źródła f (pliku) do zmiennej zawartość. Następnie źródło zostaje zamknięte.

Funkcja open()

Podstawowa składnia tej funkcji jest następująca:

```
file = open('sciezka_do_pliku', 'tryb', encoding=None)
```

- 'r' – tryb odczytu (domyślny), pozwala na czytanie zawartości pliku.
- 'r+' – tryb odczytu i modyfikacji
- 'w' – tryb zapisu, tworzy nowy plik lub nadpisuje istniejący.
- 'a' – tryb dopisywania, dodaje nowe dane na końcu pliku bez nadpisywania istniejącej zawartości.
- 'x' – otwarte do wyłącznego tworzenia, kończy się niepowodzeniem, jeśli plik już istnieje
- 'x+' – otwarte do wyłącznego tworzenia z możliwością odczytywania

Każdy z tych trybów (oprócz ‚b') domyślnie otwiera plik do zapisu jako plik tekstowy. Jeżeli chcemy zapisywać plik formie binarnej musimy uzupełnić atrybut otwarcia o literę 'b', na przykład

Funkcja open()

- 'b' – tryb binarny, używany do pracy z plikami binarnymi.

Zwykle pliki są otwierane w trybie tekstowym, co oznacza, ze odczytujesz i zapisujesz z i do pliku ciągi znaków, które są zakodowane w określonym kodowaniu. Jeśli kodowanie nie jest określone, domyślne jest zależne od platformy. Ponieważ UTF-8 jest współczesnym standardem de facto, zaleca się encoding="utf-8", chyba że wiesz, że musisz użyć innego kodowania. Dodanie „b" do trybu otwiera plik w trybie binarnym. Dane w trybie binarnym są odczytywane i zapisywane jako obiekty bajtów. Nie możesz określić kodowania podczas otwierania pliku w trybie binarnym.

W trybie tekstowym, domyślnym ustawieniem podczas czytania jest konwersja zakończeń wierszy specyficznych dla platformy (`\n`) w systemie Unix, `\r`, `\n` w systemie Windows) na samo `\n`. Podczas pisania w trybie tekstowym, domyślnym ustawieniem jest konwersja wystąpień `\n` z powrotem na zakończenia wierszy specyficzne dla platformy. Ta modyfikacja danych pliku w tle jest w porządku dla plików tekstowych, ale uszkodzi dane binarne, takie jak te w plikach JPEG lub EXE. **Należy zachować szczególną ostrożność podczas korzystania z trybu binarnego podczas czytania i zapisywania takich plików.**

Metoda close()

Metoda close() zamyka otwarty plik.
Zawsze powinieneś zamykać swoje pliki! W niektórych przypadkach, z powodu buforowania, zmiany wprowadzone do pliku mogą nie być widoczne, dopóki nie zamkniesz pliku.

with

Używanie with podczas pracy z obiektami plików należy do dobrych praktyk. Zaletą tego podejścia jest to, że plik jest prawidłowo zamykany po zakończeniu jego bloku, nawet jeśli w pewnym momencie zostanie zgłoszony wyjątek. Użycie with jest również znacznie krótsze niż pisanie równoważnych bloków try - finally:

```
with open('plik.txt', encoding="utf-8") as f:
    read_data = f.read()
    print(read_data)

# Możemy sprawdzić, że plik został automatycznie zamknięty.
print(f.closed)#True
```

Uwaga! Wywołanie f.write() bez użycia with lub f.close() może spowodować, że argumenty f.write() nie zostaną w pełni zapisane na dysku, nawet jeśli program zakończy się pomyślnie.

Po zamknięciu obiektu pliku, zarówno przez instrukcję with, jak i f.close(), wszystkie próby użycia obiektu pliku automatycznie się nie powiodą.

Metody obiektów plików

Zakładamy, że obiekt pliku o nazwie f został już utworzony.

Aby odczytać zawartość pliku, należy wywołać polecenie f.read(size), które odczytuje pewną ilość danych i zwraca je jako ciąg znaków (w trybie tekstowym) lub obiekt bajtowy (w trybie binarnym). size jest opcjonalnym argumentem numerycznym. Gdy size jest pominięty lub ujemny, zostanie odczytana i zwrócona cała zawartość pliku. W przeciwnym razie odczytane i zwrócone zostanie co najwyżej size znaków (w trybie tekstowym) lub size bajtów (w trybie binarnym). Jeśli został osiągnięty koniec pliku, f.read() zwróci pusty ciąg znaków (''').

Wczytywanie po linii

W przypadku dużych plików lepiej zastosować procedurę czytania linia po linii. Czytanie linia po linii jest też operacją stosowaną, gdy chcemy odczytać z pliku konkretne linie:

```
f = open("plik.txt")
text = f.readline()
print(text, end="*\n")
text = f.readline()
print(text, end="*\n")
text = f.readline()
print(text, end="*\n")
f.close()
```

```
abrakadabra
*
abra
*
kadabra*
```

Wczytywanie po linii – to samo z with

```
with open("plik.txt") as f:
    text = f.readline()
    print(text, end="*\n")
    text = f.readline()
    print(text, end="*\n")
    text = f.readline()
    print(text, end="*\n")
```

```
abrakadabra
*
abra
*
kadabra*
```

Wczytywanie linii do końca pliku w listą

```
f = open("plik.txt")
text = f.readlines()
print(text)
f.close()
```

['abrakadabra\n', 'abra\n', 'kadabra']

Lub przy pomocy pętli:

```
f = open("plik.txt")
text = 1
while text:
    text = f.readline()
    print(text, end='*')

f.close()
```

abrakadabra
*abra
*kadabra**

Wczytywanie linii do końca pliku w listą, to samo z with

```
with open("plik.txt") as f:
    text = f.readlines()

print(text)
```

['abrakadabra\n', 'abra\n', 'kadabra']

Lub przy pomocy pętli:

```
with open("plik.txt") as f:
    text = 1
    while text:
        text = f.readline()
        print(text, end='*')
```

abrakadabra
*abra
*kadabra**

Jeszcze wczytywanie po linii w pętli

```
with open('plik.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

abrakadabra
abra
kadabra

Funkcja strip() usuwa znaki białe (takie jak spacje i nowe linie) z początku i końca każdej linii, co jest przydatne do czyszczenia danych. Wszystkie użyte funkcje są funkcjami systemowymi wbudowanymi w interpreter języka, zatem nie musimy importować żadnych dodatkowych bibliotek.

Dalej zawsze będziemy używać with, a nie open()–close()

Zapisywanie danych do pliku

Zapis danych do pliku w Pythonie jest równie prosty jak ich odczyt. Aby zapisać dane do pliku, musimy otworzyć plik w trybie zapisu ('w') lub dopisywania ('a'). Warto nie mylić tych dwóch trybów – w pierwszym z nich jeżeli wskazany plik istnieje jego zawartość zostanie usunięta i zastąpiona przez nową. W drugim przypadku nowe dane zostaną dodane na końcu pliku. W obu przypadkach, jeżeli wskazany plik nie istnieje, zostanie on utworzony. Poniżej przykładu kodu w obu trybach:

- ```
  with open('output.txt', 'w') as file:
      file.write('To jest przykładowy tekst zapisany do pliku.')
  ```
- ```
  with open('output.txt', 'a') as file:
      file.write('\nDodajemy nową linię tekstu.')
  ```

Zapisywanie danych do pliku 'x'

Jest jeszcze jeden warty uwagi a mało znany tryb zwany Exclusive creation. Za jego pomocą otworzymy plik do zapisu, ale tylko jeśli plik nie istnieje. Jeśli plik istnieje, Python zgłosi błąd FileExistsError.

- ```
  with open('output1.txt', 'x') as file:
      file.write('To jest przykladowy tekst zapisany do pliku.')
  ```
- ```
  with open('plik.txt', 'x') as file:
      file.write('To jest przykladowy tekst zapisany do pliku.')
  ```

writelines

Metoda writelines() zapisuje elementy listy do pliku.
Miejsce, w którym zostaną wstawione teksty, zależy od trybu pliku i pozycji strumienia.

- 'a' – Teksty zostaną wstawione w bieżącej pozycji strumienia pliku, domyślnie na końcu pliku.
- 'w' – Plik zostanie opróżniony przed wstawieniem tekstów w bieżącej pozycji strumienia pliku, domyślnie 0.

```
with open("plik.txt", "a") as f:
    f.writelines(["See you soon!", "Over and out."])


#open and read the file after the appending:
with  open("plik.txt", "r") as f:
    print(f.read())
```

Metoda seek()

Metoda seek() ustawia bieżącą pozycję pliku w strumieniu plików.

Metoda seek() zwraca również nową pozycję.

- ```
  with open("plik.txt", "r") as f:
      a = f.seek(5)
      print(a)
      print(f.readline())
  ```

  ```
  5
  adabra
  ```

Metoda seekable() zwraca True, jeśli plik jest wyszukiwalny, False, jeśli nie. Plik jest wyszukiwalny, jeśli umożliwia dostęp do strumienia plików, tzn metoda seek().
  ```

Odczyt i zapis

Istnieją też tryby mieszane, które łączą tryby tekstowe z trybami binarnymi oraz operacje zapisu i odczytu. Dzięki nim możemy wykonywać bardziej zaawansowane operacje na plikach, które wymagają zarówno czytania, jak i modyfikowania zawartości pliku bez konieczności zamykania i ponownego otwierania go w innym trybie. Tryby mieszane są szczególnie przydatne w scenariuszach, gdzie dane muszą być dynamicznie aktualizowane lub weryfikowane, a następnie zapisywane z powrotem do tego samego pliku. Tryby mieszane poznamy po tym, że w atrybutach ich otwarcia znajduje się znak '+'. Przykładowo:

```
with open('plik.txt', 'r+') as file:
    content = file.read()
    file.write('Dopisana zawartosc.')
```

Powyższe kod otwiera plik do odczytu i zapisu. Jeśli plik nie istnieje, Python zgłosi błąd FileNotFoundError.

Po odczytywaniu wskaźnik jest na końcu plika, i treść do zapisywania zapisuje się na koniec.

Gdzie się zapisujemy?

Porównaj z poprzednim:

- Treść do zapisywania zapisuje się na początku pliku.

```
with open('plik.txt', 'r+') as file:
    file.write('Dopisana zawartosc.')
```

-
```
with open('plik.txt', 'r+') as file:
    file.seek(5)
    file.write('Dopisana zawartosc.')
```

Metoda seek() ustawia bieżącą pozycję pliku w strumieniu plików.

Porownaj

- ```
  with open("plik.txt", "r+") as f:
      a = f.seek(5)
      print(a)
      f.write('See you soon!')
      f.seek(0)
      print(f.read())
  ```

  ```
  5
  abrakSee you soon!adabra
  ```

- ```
  with open("plik.txt", "r+") as f:
      a = f.seek(5)
      print(a)
      print(f.write('See you soon!'))
      print(f.read())
      f.seek(0)
      print(f.read())
  ```

  ```
  5
  13
  adabra
  abrakSee you soon!adabra
  ```

Inne metody

- Metoda tell() zwraca bieżącą pozycję pliku w strumieniu plików.
- Metoda readable() zwraca True, jeśli plik jest czytelny, lub False w przeciwnym razie.
- Metoda writable() zwraca True, jeśli do pliku można zapisywać, lub False w przeciwnym razie.

truncate()

- Metoda truncate() zmienia rozmiar pliku do podanej liczby bajtów.

```
with open("plik.txt", "a") as f:
    f.truncate(10)

#open and read the file after the truncate:
with open("plik.txt", "r") as f:
    print(f.read())
```

abrakadabr

Przykład

Źródło:
`https://www.flynerd.pl/2018/01/python-metody-typu-string.html`

Wyobraź sobie, że jesteś bioinformatykiem i otrzymujesz kod genetyczny do analizy w pliku tekstowym.
Kod DNA składa się z 4 zasad azotowych: adeniny(A), cytozyny(D), guaniny(G), tyminy(T). Idealny kod DNA wygląda następująco:

TGCACGATCATGTCTACTATCCTCTCTATGGTGGGGTT...

Zdarza się, jednak, że kod zawiera małe jak i duże litery. Kolejny problem to maszyny sekwencjonujące nie są wolne od błędów. W zależności od maszyny błędy sekwencjonowania mogą zostać zamienione na znak – czy literę N.

- wczytaj plik
- policz ile razy występuje w kodzie każda zasada azotowa - adenina, cytozyna, guanina, tymina.

Przykład: rozwiązanie

```
with open("DNA.txt", "r") as f:
    seq0 = f.read()

seq = seq0.strip().upper()
print(seq)
n = len(seq)
occurences ={}
for x in 'ACGT':
    occurences[x] = seq.count(x)

print(occurences)
```

Przykład dalej

W dokumentacji znajduje się następujący zapis:
gdy jakość sekwencji nie pozwala dokładnie odczytać rodzaju zasady azotowej wstawiany jest znak „-” Natomiast, gdy laser sczytujący ześlizgnie się, wstawiane są litery „N”, jednocześnie ostatnia wartość zasady jest ponownie odczytywana bez ubytku zasady w tym miejscu.
Co za przydatna informacja!

- Oczyść DNA z błędów typu N.
- Policz wystąpienia sekwencji GAGA
- Znajdź miejsce (indeks) w łańcuchu, gdzie występuje 7 guanin z rzędu
- Znajdź miejsce (indeks) , gdzie od końca łańcucha występuje 6 cytozyn
- Policz ile razy w kodzie pojawiła się sekwencja CTGAAA
- W sekwencji CTGAAA czasem mutuje ostania litera A, wówczas jakość ostatniej litery może być wątplia. Ile sekwencji znajdziesz, jeśli weźmiesz pod uwagę wątpliwą, ostatnią adeninę?
- Na podstawie czystej nici utwórz odpowiadającą jej nić RNA (nić RNA w miejscu tyminy będzie mieć uracyl (U)). Nic RNA zapisz do nowego pliku RNA.txt

Przykład: rozwiązanie dalej

```
with open("DNA.txt", "r") as f:with open("DNA.txt", "r") as f:
    seq0 = f.read()

seq = seq0.strip().upper()
print(seq)
n = len(seq)
DNA = seq.replace("N", "")
GAGA = DNA.count("GAGA")
print("Liczba wystąpień sekw. GAGA", GAGA)

CTGAAA = DNA.count("CTGAAA")
print("Liczba wystąpień sekw. CTGAAA", CTGAAA)

CTGAA_ = DNA.count("CTGAA-")
print("Liczba wystąpień sekw. CTGAAA i CTGAA-", CTGAAA + CTGAA_)

RNA = DNA.replace("T", "U")
with open("RNA.txt", "w") as f:
    f.write(RNA)
```

Czytanie i przetwarzanie plików tabelarycznych

Ponieważ python wczytuje dane w postaci pojedynczej zmiennej tekstowej, jeżeli wczytane dane zamierzamy poddać obliczeniom, należy je odpowiednio przekształcić, najczęściej do postaci tabelarycznej oraz zmodyfikować typy danych z tekstowych do numerycznych. Do tego celu służą poznane już funkcje split() oraz funkcje konwersji zmiennych. Procedura czytania danych jest następująca:

- W pierwszysm kroku nawiązywane jest połączenie z plikiem (zawartość zostaje wyświetlona)
- Dane zostają podzielone na listę łańcuchów tekstowych, względem znaku "\ną następnie każda linia zostaje podzielona na listy wewnętrzne na podstawie przecinka lub srednika (separator w csv). Wynik zostaje wyświetlony jako lista zagnieżdżona
- Połączenie zosaje zamknięte
- Z listy zostaje usunięta 1 linia (nagłówek)
- Z listy zostaje usunięta ostatnia pusta linia (z reguły występuje w plikach tekstowych)

Punkty 1-3 łącząc się przy pomocy with

Czytanie i przetwarzanie plików tabelarycznych

```
f = open("tab.csv") #1
dane = f.read() #
print(dane)
dane = dane.split('\n') #2 podziel po liniach
dane = [l.split(';') for l in dane] #2 podziel po wierszach
print(dane)
f.close() #3
header = dane.pop(0) #4 nagłówek
dane.pop() #5 usunięcie ostatniego, pustego wiersza
```

```
Ulica;Nomer domu;Kod;Miejscowosc
Pasikonika ;20;81-098;Lubowo
Biedronki;15;81-789;Bobowo
Zuczka;10;61-874;Rabowo
Gasienicy;33;41-879;Warbowo
[['Ulica', 'Nomer domu', 'Kod', 'Miejscowosc'],
['Pasikonika ', '20', '81-098', 'Lubowo'],
['Biedronki', '15', '81-789', 'Bobowo'],
['Zuczka', '10', '61-874', 'Rabowo'],
['Gasienicy', '33', '41-879', 'Warbowo'], ['']]
```

Czytanie i przetwarzanie plików tabelarycznych:with

```
with open("tab.csv") as f:#1
    dane = f.read() #

print(dane)
dane = dane.split('\n') #2 podziel po liniach
dane = [l.split(';') for l in dane] #2 podziel po wierszach
print(dane)
header = dane.pop(0) #4 nagłówek
dane.pop() #5 usunięcie ostatniego, pustego wiersza
```

```
Ulica;Nomer domu;Kod;Miejscowosc
Pasikonika ;20;81-098;Lubowo
Biedronki;15;81-789;Bobowo
Zuczka;10;61-874;Rabowo
Gasienicy;33;41-879;Warbowo
[['Ulica', 'Nomer domu', 'Kod', 'Miejscowosc'],
['Pasikonika ', '20', '81-098', 'Lubowo'],
['Biedronki', '15', '81-789', 'Bobowo'],
['Zuczka', '10', '61-874', 'Rabowo'],
['Gasienicy', '33', '41-879', 'Warbowo'], ['']]
```

Transponowanie danych

Dane są przechowywane w postaci listy list, gdzie każda lista zagnieżdżona to pojedynczy wiersz składający się z danych różnego typu. Ponieważ operowanie na wierszach zawierających dane różnego typu nie jest wygodne, można przekształcić listę zagnieżdżoną do postaci kolumnowej (dokonać transpozycji) przy pomocy funkcji zip(), którą omówimy za chwile. Dane przekazujemy z * czyli rozwijamy przekazaną listę do postaci: dane[0], dane[1],...,dane[n-1]

```
dane = zip(*dane)
trans = list(dane)
print(trans)
```

[('Pasikonika ', 'Biedronki', 'Zuczka', 'Gasienicy'), ('20', '15', '10', '33'), ('81-098', '81-789', '61-874', '41-879'), ('Lubowo', 'Bobowo', 'Rabowo', 'Warbowo')]

Przekształcenie typów danych

```
dane = zip(*dane)
trans = list(dane)
print(trans)
```

[('Pasikonika ', 'Biedronki', 'Zuczka', 'Gasienicy'), ('20', '15', '10', '33'), ('81-098', '81-789', '61-874', '41-879'), ('Lubowo', 'Bobowo', 'Rabowo', 'Warbowo')]

W ostatnim kroku możemy przekształcić 2 i 3 linię do typu integer.

```
trans[1] = tuple(int(x) for x in trans[1])
print(trans)

trans[2] = tuple(int(x[:2]+x[3:]) for x in trans[2])
print(trans)
```

Klasy

Pierwszym krokiem tworzenia własnych obiektów jest stworzenie klasy, która jest czymś w rodzaju projektu, na podstawie którego tworzone są obiekty, które są egzemplarzami klasy. Obiekty mogą przechowywać dane, a także wykonywać kod, który jest umieszczony w metodach. Metody są funkcjami, które są częściami klasy.
Klasa opisuje pewien typ obiektów. Jest więc czymś w rodzaju "projektu" na podstawie którego tworzy się konkretne obiekty, które są osobnymi bytami. W klasie możemy zdefiniować przechowywane przez obiekty rodzaje informacji, a także operacje, które obiekty będą wykonywać. Najpierw więc tworzymy projekt (klasę), a potem na jej podstawie konkretne obiekty.
Klasy można umieszczać w osobnych plikach (modułach), zwłaszcza jeśli są bardzo rozbudowane, ale wiele klas może być także umieszczonych w jednym module.

`https://www.w3schools.com/python/python_classes.asp`
`https://kt.academy/pl/article/py-klasy`
`https://ggoralski.gitlab.io/python-wprowadzenie/czesc_i/15-klasy_i_obiekty/`

Pierwszy przykład

Zacznijmy od najprostszej opcji, czyli pustej klasy. Taka klasa nic nie będzie zawierać, niemniej będzie miała swoją nazwę i będziemy mogli przy jej pomocy stworzyć obiekt

```
class Cookie:
    pass


cookie = Cookie()
```

Przy nazywaniu klas możemy używać tych samych znaków co w przypadku zmiennych i funkcji: małych i dużych liter oraz znaku podkreślenia _. Konwencja nazewnicza jest jednak inna. Dla funkcji i zmiennych używaliśmy `snake_case`. W przypadku klas używamy `PascalCase` (lub `UpperCamelCase`), czyli każde słowo zaczynamy wielką literą, nie używamy spacji ani znaków podkreślenia.

Zmienne obiektu 1

Do obiektu możemy przypisać zmienną z określoną wartością. Taka wartość dotyczyć będzie wyłącznie tego jednego obiektu. Aby odnieść się do zmiennej w obiekcie, musimy wskazać zarówno obiekt, jak i zmienną, a oddzielamy ich nazwy kropką. Dla przykładu, aby odnieść się do zmiennej type w obiekcie cookie1, użyjemy cookie1.type. Zarówno do przypisania wartości, jak i do jej pobrania.

```
class Cookie:
    pass


cookie1 = Cookie()
cookie2 = Cookie()
cookie1.type = "Biscuit"
cookie1.color = "White"
cookie2.type = "Oreo"
print(cookie1.type)  # Biscuit
print(cookie1.color)  # White
print(cookie2.type)  # Oreo
# print(cookie2.color)
```

Konstruktor i inicjalizator 1

Nieczęsto tworzy się zmienne obiektu tak jak w powyższym przykładzie: poza klasą. Często jest to wręcz uznawane za złą praktykę. Częściej tworzy się je w obrębie metod, a zwłaszcza szczególnej metody zwanej inicjalizatorem.
Gdy tworzymy nowy obiekt, stawiamy nawias za nazwą klasy. Ten nawias to wywołanie funkcji tworzącej obiekt, zwanej konstruktorem. Funkcja ta przechodzi przez szereg kroków, niezbędnych do utworzenia obiektu, w tym między innymi woła specjalną metodę o nazwie `__init__` z naszej klasy. Ta metoda zwana jest inicjalizatorem. W jej ciele określamy, co powinno się dziać w czasie tworzenia obiektu. Najczęściej definiujemy w niej atrybuty obiektu.

```
class Cookie:
    def __init__(self, type, color):
        self.type = type
        self.color = color
#pierwszym parametrem jest odniesienie do instancji obiektu,
#na którym tę metodę wywołamy

cookie1 = Cookie("Biscuit","White")
print(cookie1.type)  # Biscuit
print(cookie1.color)  # White
```

Konstruktor i inicjalizator 2

Liczba parametrów funkcji `__init__` określa, ile argumentów powinno się znaleźć w wywołaniu konstruktora (czyli nawiasie, który stawiamy za nazwą klasy, gdy tworzymy obiekt). Jeśli więc w funkcji `__init__` dodamy parametr name, to przy tworzeniu obiektu nie możemy już zostawić pustego nawiasu. Powinniśmy podać tam argument, który posłuży jako imię. Typowym dla funkcji `__init__` jest, że spodziewa się określonych parametrów, po czym przypisuje je do obiektu jako atrybuty o takiej samej nazwie.

```
class Cookie:
    def __init__(self, type, color = None):
        self.type = type
        self.color = color


cookie1 = Cookie("Biscuit","White")
cookie2 = Cookie("Oreo")
print(cookie1.type)  # Biscuit
print(cookie1.color)  # White
print(cookie2.type)  # Oreo
print(cookie2.color) #None
```

Jeszcze przykład

```
class Player:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
        self.full_name = f"{name} {surname}"
        self.points = 0


player = Player("Michał", "Mazur")
print(player.name)  # Michał
print(player.surname)  # Mazur
print(player.full_name)  # Michał Mazur
print(player.points)  # 0
```

Metody

Wewnątrz klas możemy definiować funkcje. Takie funkcje nazywane są metodami. Definiujemy je w ciele klasy, a ich pierwszym parametrem jest odniesienie do instancji obiektu, na którym tę metodę wywołamy. Parametr ten powinno nazywać się `self`. Gdy wywołujemy metodę, zaczynamy od obiektu, następnie stawiamy kropkę, nazwę metody i nawias z argumentami.

```
class Position:
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y
    def step_right(self):
        self.x += 1.0
    def move_up(self, value):
        self.y += value

pos = Position()
pos.step_right()
print(pos.x)  # 1.0
pos.move_up(6.0)
print(pos.y)  # 6.0
pos.move_up(3.0)
print(pos.y)  # 9.0
```

Obiekty i zmienne 1

Każdy obiekt jest osobnym bytem. To, że wyglądają podobnie, nie znaczy, że mają na siebie wpływ. Dlatego też w poniższym przykładzie zmiana name w obiekcie user1 nie będzie miała żadnego wpływu na user2.

```
class User:
    def __init__(self, name):
        self.name = name


user1 = User("Rafał")
user2 = User("Rafał")

print(user1.name)  # Rafał
print(user2.name)  # Rafał

user1.name = "Bartek"

print(user1.name)  # Bartek
print(user2.name)  # Rafał
```

Obiekty i zmienne 1

Każdy obiekt jest osobnym bytem. To, że wyglądają podobnie, nie znaczy, że mają na siebie wpływ. Dlatego też w poniższym przykładzie zmiana name w obiekcie user1 nie będzie miała żadnego wpływu na user2.

```
class User:
    def __init__(self, name):
        self.name = name


user1 = User("Rafał")
user2 = User("Rafał")

print(user1.name)  # Rafał
print(user2.name)  # Rafał

user1.name = "Bartek"

print(user1.name)  # Bartek
print(user2.name)  # Rafał
```

Obiekty i zmienne 2

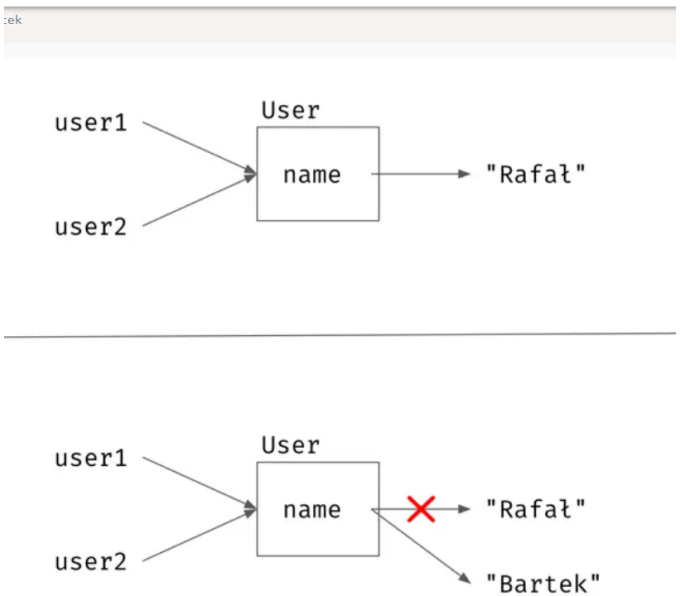
Z drugiej strony, jeśli mamy dwie zmienne wskazujące na jeden obiekt, to możemy go zmienić przy użyciu dowolnej z nich. Po takim zabiegu, wartości dla obu zmiennych ulegną zmianie, bo przecież przekształcone zostało coś, na co obydwie wskazują.

```
user1 = User("Rafał")
user2 = user1

print(user1.name)
# Rafał
print(user2.name)
# Rafał

user1.name = "Bartek"

print(user1.name)
# Bartek
print(user2.name)
# Bartek
```



Rysunek: Źródło: https://kt.academy/pl/article/py-klasy

Obiekty i zmienne 3

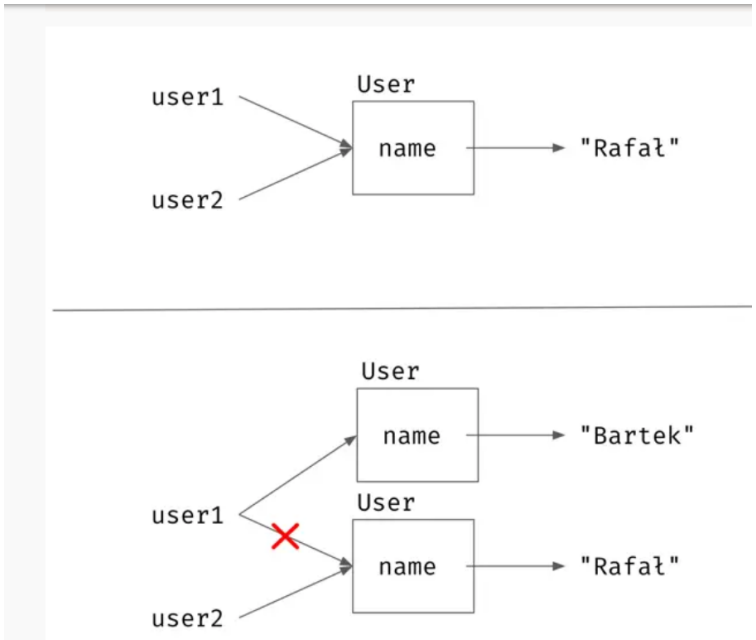
Dwie zmienne wskazują na ten sam obiekt i właściwość tego obiektu zmienia wartość.
Warto porównać to z przykładem, gdy dwa obiekty pokazywały na tą samą wartość, a potem zmieniło się to, na co jedna z tych zmiennych wskazuje.
Wynik będzie inny.

```
user1 = User("Rafał")
user2 = user1

print(user1.name)
# Rafał
print(user2.name)
# Rafał

user1 = User("Bartek")

print(user1.name)
# Bartek
print(user2.name)
# Rafał
```



Rysunek: Źródło: https://kt.academy/pl/article/py-klasy

Metoda copy()

Dwie zmienne wskazują na ten sam obiekt i właściwość tego obiektu zmienia wartość.

```
class User:
    def __init__(self, name):
        self.name = name

    def copy(self):
        return User(self.name)

user1 = User("Rafał")
user2 = user1.copy()
user1.name = "Bartek"
print(user1.name)  # Bartek
print(user2.name)  # Rafał
user2.name = "Marek"
print(user1.name)  # Bartek
print(user2.name)  # Marek
```

Metoda __str__()

```
class User:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'Username:{self.name}'

user1 = User("Rafał")
print(user1)
```

Przykład: klasa liczb rzymskich

Funkcja pierwsza:

```
def to_arabic(number):
    roman_numerals = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100,
                                    'D': 500, 'M': 1000}
    # 4 (IV) and 9 (IX), 40 (XL), 90 (XC), 400 (CD) and 900 (CM)
    number = number.replace('IV', 'IIII')
    number = number.replace('IX', 'VIIII')
    number = number.replace('XL', 'XXXX')
    number = number.replace('XC', 'LXXXX')
    number = number.replace('CD', 'CCCC')
    number = number.replace('CM', 'DCCCC')
    return sum(roman_numerals[i] for i in number)

print(to_arabic('MCDLXIV'))
```

Przykład: klasa liczb rzymskich

Funkcja druga:

```
def to_roman(n):
    roman_numerals = {1000:'M', 900: 'CM', 500: 'D', 400: 'CD', 100:
                                'C', 90:'XC',50: 'L', 40:'XL',
                                10:'X', 9:'IX', 5: 'V' ,4:'IV', 1: 'I'}
    s = ''
    for i in roman_numerals:
        div = n // i
        n %= i
        while div:
            s += roman_numerals[i]
            div -= 1
    return s

print(to_roman(1464))
```

Przykład: konstruktor

```
class Roman:
    def __init__(self, n):
        #two constructors are not allowed
        if isinstance(n, str):
            self.roman = n
            roman_numerals = {'I': 1, 'V': 5, 'X': 10, 'L': 50,
                              'C': 100, 'D': 500, 'M': 1000}

            n = n.replace('IV', 'IIII')
            n = n.replace('IX', 'VIIII')
            n = n.replace('XL', 'XXXX')
            n = n.replace('XC', 'LXXXX')
            n = n.replace('CD', 'CCCC')
            n = n.replace('CM', 'DCCCC')
            self.arabic = sum(roman_numerals[i] for i in n)
```

Przykład: konstruktor dalej

```
        elif isinstance(n, int):
            if n<=0:
                raise ValueError("Roman number should be positive")
            self.arabic = n
            romanian_numerals = {1000: 'M', 900: 'CM', 500: 'D',
                       400: 'CD', 100: 'C', 90: 'XC', 50: 'L', 40: 'XL'
                       10: 'X', 9: 'IX', 5: 'V', 4: 'IV', 1: 'I'}
            s = ''
            for i in romanian_numerals:
                div = n // i
                n %= i
                while div:
                    s += romanian_numerals[i]
                    div -= 1
            self.roman = s


    def __str__(self):
        return self.roman

print(Roman(10))#X
```

Przykład: funkcja

```
class Roman:
    def __init__(self, n):
        ...
    def __str__(self):
        ...

r = Roman("XVI")
print(r.arabic)#16
```

Przykład: dodawanie i mnożenie

```
class Roman:
    def __init__(self, n):
        ...
    def __str__(self):
        ...

    def __add__(self, other):
        return Roman(self.arabic+other.arabic)

    def __mul__(self, other):
        return Roman(self.arabic*other.arabic)
r = Roman("XVI")
p = Roman("X")
print(r+p)#XXVI
print(r*p)#CLX
```

Przykład: odejmowanie i dzielenie

```
class Roman:
    ....

    def __sub__(self, other):
        return Roman(self.arabic-other.arabic)

    def __truediv__(self,other):
        if self.arabic%other.arabic == 0:
            return Roman(self.arabic//other.arabic)
        raise ValueError ("Roman number should be integer")

print(Roman('X')-Roman('IX'))#I
print(Roman('XX')/Roman('X'))#II

print(Roman('X')-Roman('X'))#ValueError
print(Roman('VIII')/Roman('X'))#ValueError
```

Dziedziczenie(inheritance)

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

 class Student(Person):
  pass


x = Person("John", "Doe")
x.printname()

x = Student("Mike", "Olsen")
x.printname()
```

Dziedziczenie(inheritance)

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self, fname, lname, number):
        Person.__init__(self, fname, lname)
        #super().__init__(fname, lname)
        self.number = number


x = Person("John", "Doe")
x.printname()

x = Student("Mike", "Olsen", 1999)
x.printname()
```

Jeszcze przykład

```
class Fruit:

    def __init__(self, name, sugar_content):
        self.name = name
        self.sugar_content = sugar_content

class Apple(Fruit):

    def __init__(self):
        super().__init__("apple", 2)
```