Tytuł rozdziału

Podstawy programowania obiektowego

Uwagi wstępne

- Podstawowym celem tej części wykładu jest zaprezentowanie fascynującej i fundamentalnej koncepcji programowania zorientowanego obiektowo.
- Programowane zorientowane obiektowo nie jest tylko tajemniczą przyprawą w programowaniu grafiki 3D oraz gier wideo; jest ono powszechnie stosowane w realizowanych obecnie dużych projektach związanych z wytwarzaniem oprogramowania.
- Poznamy wybrane podstawy programowania zorientowanego obiektowo (nazywanego także krócej: programowaniem obiektowym).

Przykład wprowadzający – testowanie gry (1/5)

☞ Wyobraźmy sobie gracza komputerowego, którego zadaniem jest wylądować pojazdem kosmicznym na powierzchni pewnej planety.

- W tym celu gracz musi odpalić silniki manewrowe w celu nakierowania statku na lądowisko oraz wyhamowania do poziomu prędkości lądowania. Gra pokazuje ile paliwa pozostało oraz jaka ilość paliwa jest wymagana do wykonania poszczególnych manewrów.

☞ Załóżmy dalej, że mamy za zadanie przetestować taką grę.

- W trakcie testowania zauważamy, że często pojawia się informacja o niewystarczającej ilości paliwa potrzebnej do wykonania kluczowych manewrów, podczas gdy, w rzeczywistości, jesteśmy przekonani, że powinno być akurat tyle paliwa ile potrzeba.
- Naszym zadaniem jest wykrycie nieprawidłowości oraz wskazanie sposobu usunięcia błędu.

Przykład wprowadzający – testowanie gry (2/5)

☞ Każdy spośród zamontowanych w rakiecie dwóch zbiorników paliwa ma pojemność 1000 jednostek; wskaźnik paliwa dla każdego zbiornika pokazuje pewną wartość pomiędzy 0 a 1.0, oznaczającą jaka część paliwa pozostała w zbiorniku.

```
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```

- Kod pokazuje przykładowy test, gdzie wartość fuelNeeded wskazuje jaka część zbiornika mieszczącego 1000 jednostek jest wymagana do wykonania manewru, a wartości tank1 oraz tank2 pokazują w jakiej części oba zbiorniki są wypełnione paliwem.

- Ostatnie polecenie sprawdza, czy całkowita wartość paliwa w obu zbiornikach jest większa lub równa ilości wymaganej przez manewr.
- Zauważmy, że $\frac{36}{1000} + \frac{6}{1000} = \frac{42}{1000}$, a wymagana ilość paliwa jest równa dokładnie $\frac{42}{1000}$. Dziwi zatem to, że nasz kod daje odpowiedź, że nie ma wystarczającej do wykonania manewru ilości paliwa, skazując statek na katastrofę.
- Dopiero wyświetlenie dokładnych wartości fuelNeeded, tank1, tank2 oraz tank1 + tank2 pozwala dostrzec w czym problem.

Przykład wprowadzający – testowanie gry (3/5)

- Wyświetlając wartości fuelNeeded, tank1, tank2 oraz tank1 + tank2 z większą liczbą cyfr po przecinku otrzymujemy:

```
>>> print ('{:.18f}'.format(fuelNeeded))
0.042000000000000003
>>> print ('{:.18f}'.format(tank1))
0.035999999999999997
>>> print ('{:.19f}'.format(tank2))
0.0060000000000000001
>>> print ('{:.18f}'.format(tank1+tank2))
0.041999999999999996
```

- Pokazany przykład niedokładności numerycznej jest efektem błędu wewnętrznego, który pojawia się kiedy komputery dokonują zamiany ułamków na liczby zmiennoprzecinkowe (liczby reprezentowane jako dziesiętne zmiennoprzecinkowe).

- Przyjmując jednak, że wszystkie wielkości mierzone w naszej rakiecie są zawsze liczbami wymiernymi – tzn. ułamkami z całkowitym licznikiem i całkowitym mianownikiem – wspomniany problem niedokładności jest do uniknięcia!
  - Jak? W przypadku liczb całkowitych problem niedokładności nie istnieje. A zatem dla dowolnej liczby wymiernej możemy przechowywać odpowiednie wartości całkowite licznika i mianownika a następnie wykonywać wszelkie operacje arytmetyczne na liczbach całkowitych.

Przykład wprowadzający – testowanie gry (4/5)

- Na przykład,
  - Liczba wymierna $\frac{36}{1000}$ może być zapamiętana jako para liczb całkowitych 36 and 1000 bez przekształcania jej na liczbę zmiennoprzecinkową.
  - Aby obliczyć $\frac{36}{1000} + \frac{6}{1000}$ wystarczy wskazać $36 + 6 = 42$ jako wartość licznika oraz 1000 jako wartość mianownika.
  - Porównanie uzyskanego wyniku z wartością fuelNeeded wynoszącą $\frac{42}{1000}$ wymaga osobnego porównania liczników i mianowników, to jest porównania ze sobą liczb całkowitych, które nie jest narażone na błędy numeryczne.
- A zatem, byłoby wspaniale, jeśliby w Pythonie była możliwość operowania na liczbach wymiernych, jako parach liczb całkowitych.
  - Innymi słowy, chcielibyśmy mieć typ danych „liczby wymierne”, nawiązujący do istniejących języku Python typów danych m.in. takich jak: integer, string czy list.
  - Ponadto byłoby świetnie móc wykonywać operacje arytmetyczne i porównania na liczbach wymiernych z łatwością porównywalną do tej, jaka ma miejsce dla liczb całkowitych.

Przykład wprowadzający – testowanie gry (5/5)

- Twórcy języka Python z pewnością nie mogli przewidzieć wszelkich możliwych typów danych, które będą potrzebne użytkownikom.
  - Zamiast tego Python (podobnie jak wiele innych języków) pozwala programiście w wygodny sposób definiować własne nowe typy a następnie używać je z łatwością porównywalną do tej z jaką używamy typów wbudowanych, takich jak: liczby całkowite, łańcuchy znaków czy listy.
- Ta możliwość definiowania nowych typów danych jest określana jako programowanie obiektowe.
  - Można niekiedy spotkać skrót OOP odwołujący się do angielskiego terminu *object-oriented programming* co oznacza programowanie zorientowane na obiekty.
  - Programowanie obiektowe jest zasadniczym tematem tej części naszego wykładu.

Klasa **Rational** (1/5)

- Zdefiniujmy zatem typ liczby wymiernej.
  - W tym celu zbudujemy w środowisku Python „fabrykę”, wytwarzającą liczby wymierne. Taką fabrykę określa się pojęciem klasa, a wygląda ona następująco:

```
class Rational:
  def __init__(self, num, denom):
    self.numerator = num
    self.denominator = denom
```

- Nie należy w tej chwili przejmować się nieco dziwną składnią poleceń występujących w tym kodzie; wrócimy do tego za jakiś czas, kiedy bliżej przyjrzymy się szczegółom.
  - Teraz najważniejsze jest to, że skoro napisaliśmy klasę Rational (oraz zapamiętaliśmy ją w pliku o takiej samej nazwie z rozszerzeniem .py, w tym przypadku Rational.py) to możemy „produkować” – lub używając przyjętego terminu inicjalizować (*instantiate*) – nowe liczby wymierne, w ilości jakiej tylko zechcemy.

Klasa **Rational** (2/5)

☞ Oto przykład wywołania wspomnianej „fabryki” celem inicjalizacji dwóch liczb wymiernych $\frac{36}{1000}$ oraz $\frac{6}{1000}$:

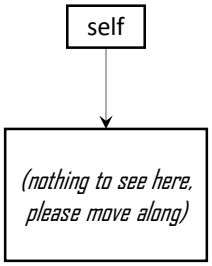
```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

☞ Co tutaj mamy? Gdy Python napotyka instrukcję

```
r1 = Rational(36, 1000)
```

wykonuje dwie czynności:

- Po pierwsze inicjalizuje pusty obiekt, który będziemy nazywać self. Właściwie self jest referencją (wskaźnikiem) na pusty obiekt co zilustrowano na rysunku obok.
- Następnie Python przegląda klasę Rational w poszukiwaniu funkcji o nazwie __init__ (zwróćmy uwagę na dwa znaki podkreślenia przed i po słowie „init”). Jest to trochę dziwna nazwa ale zgodna z przyjętą konwencją.




```
class Rational:
  def __init__(self, num, denom):
    self.numerator = num
    self.denominator = denom
```


Klasa **Rational** (3/5)

```
class Rational:
  def __init__(self, num, denom):
    self.numerator = num
    self.denominator = denom
```

☞ Zauważmy, że z definicji funkcji __init__ wynika, że funkcja pobiera trzy argumenty, gdy tymczasem w linii r1 = Rational(36, 1000) podano tylko dwa.

- To trochę dziwne, ale – jak być może się domyślamy – pierwszy argument zostaje wstawiony automatycznie przez system Python i jest referencją do nowego pustego obiektu self który właśnie został zainicjalizowany.

☞ Funkcja __init__ pobiera referencję self do naszego nowego pustego obiektu i przechodzi do umieszczenia danych w tym obiekcie. Są to wartości 36 oraz 1000, które zostają przekazane do __init__ odpowiednio jako num oraz denom.

- Dalej, gdy funkcja __init__ wykonuje polecenie self.numerator = num , to nakazuje „iść do obiektu wskazanego przez self, wydzielić dla niego zmienną o nazwie numerator oraz nadać tej zmiennej wartość, która została przekazana jako num.” Analogicznie polecenie self.denominator = denom mówi: „idź do obiektu wskazanego przez self, wydziel dla niego zmienną o nazwie denominator oraz nadaj tej zmiennej wartość, która została przekazana jako denom.” Zauważmy, że nazwy num, denom, numerator oraz denominator nie są z góry ustalone – zostały one wybrane przez nas.
- Zmienne numerator oraz denominator w funkcji __init__ są nazywane atrybutami klasy Rational. Klasa może mieć tyle atrybutów ile dla niej zdefiniowano. Wydaje się zrozumiałe, że klasa liczb wymiernych powinna mieć przynajmniej te dwa argumenty!

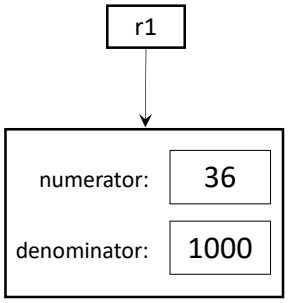
Klasa **Rational** (4/5)

- Ostatnią czynnością wykonaną w linii r1 = Rational(36, 1000) jest nadanie zmiennej r1 wartości referencji do obiektu, który właśnie został przez Python utworzony, a przez nas inicjalizowany.

☞ Zawartości liczb wymiernych można podglądać następująco:

```
>>> r1.numerator
36
```

- W definicji funkcji __init__ także użyliśmy „kropki” w poleceniu self.numerator = num. Kropka pełniła tam tę samą rolę „mówiąc”: „idź do obiektu i zajmij się atrybutem o nazwie numerator”. Rysunek obok pokazuje aktualną strukturę.
- Zwrócimy jeszcze uwagę, że wartości numerator and denominator pokazano tak, jak gdyby stanowiły one zawartość odnośnych zmiennych. W rzeczywistości rozważane wartości całkowite mogą znajdować się w zupełnie innym miejscu pamięci, podczas gdy zmienne przechowują referencje do tych wartości.



Klasa **Rational** (5/5)

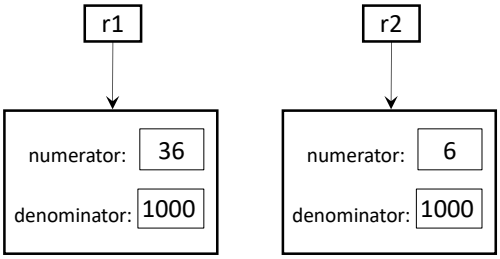
☞ W poprzednim przykładzie „fabryka” Rational została „wywołana” dwukrotnie celem inicjalizacji dwóch różnych liczb wymiernych:

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

- Pierwsze wywołanie Rational(36, 1000) inicjalizuje liczbę wymierną, self, z licznikiem 36 oraz mianownikiem 1000. Następnie wartość referencji do tego obiektu zostaje nadana zmiennej r1.

☞ Natomiast polecenie r2.numerator mówi, „idź do obiektu o nazwie r2 i zajmij się atrybutem o nazwie numerator.”

- Należy pamiętać, że skoro r1 i r2 wskazują na dwa różne obiekty, to każdy z nich ma swój „własny” numerator oraz denominator. Pokazano to rysunkach obok.



Dodawanie liczb klasy **Rational** (1/5)

- Podsumujmy, to, co udało się poznać do tej pory.
  - Po pierwsze, zdefiniowaliśmy „fabrykę” – oficjalnie określaną pojęciem klasa – o nazwie Rational. Klasa Rational stanowi (opisuje) szablon do wytwarzania danych nowego typu.
  - Wspomniana fabryka może teraz zostać użyta do inicjalizacji zbioru wystąpień – oficjalnie nazywanych obiektami – danych tego typu.
  - Każdy obiekt ma swoje własne zmienne – lub „atrybuty” – w naszym przykładzie to numerator oraz denominator, które z kolei mają własne wartości.
- Wygląda to efektownie, ale czy załatwia sprawę?
  - Pamiętamy, że powodem, dla którego zdefiniowano liczby Rational było znalezienie sposobu operowania (dodawania, porównywania, itd.) na liczbach wymiernych bez potrzeby przekształcania ich na dziedzinę zmiennoprzecinkową, gdzie błędy numeryczne powodują kłopoty.
  - Wbudowane typy danych języka Python (takie jak liczby całkowite, liczby zmiennoprzecinkowe oraz łańcuchy znaków) mogą być dodawane, porównywane itp. Chcielibyśmy wyposażyć w takie możliwości także liczby klasy Rational!

Dodawanie liczb klasy **Rational** (2/5)

- Zaczniemy od tego, że klasę Rational uzupełnimy o funkcję, która umożliwi dodawanie liczb klasy Rational, a zwracana suma będzie także liczbą klasy Rational.
  - Funkcja zdefiniowana w ramach klasy nosi specjalną i efektownie brzmiącą nazwę – jest nazywana metodą tej klasy. Ponadto metoda __init__ jest nazywana konstruktorem.
- Nasza metoda add realizująca dodawanie w klasie Rational będzie używana następująco:

```
>>> r1.add(r2)
```

  - W efekcie powinna być zwracana liczba Rational, stanowiąca wynik dodawania r1 oraz r2. A zatem powinien być możliwy zapis:

```
>>> r3 = r1.add(r2)
```

  - Teraz r3 wskazuje na nową liczbę Rational zwróconą przez metodę add. Użyta składnia może na początku wydać się trochę udziwniona, jednak warto wykazać cierpliwość; już wkrótce okaże się, że ma ona sens.

Dodawanie liczb klasy **Rational** (3/5)

☞ Napiszmy zatem metodę add!

- Jeśli $r1 = \frac{a}{b}$ oraz $r2 = \frac{c}{d}$ to $r1 + r2 = \frac{ad+bc}{bd}$. Zauważamy wprawdzie, że uzyskany w wyniku ułamek, być może, podlega uproszczeniu przez podzielenie przez czynniki wspólne dla licznika i mianownika, jednak nie zajmujemy się teraz tą kwestią.
- Oto klasa Rational wraz z nową - prosto spod igły - metodą add:

```
class Rational:
  def __init__(self, num, denom):
    self.numerator = num
    self.denominator = denom

  def add(self, other):
    newNumerator = self.numerator * other.denominator +
            self.denominator * other.numerator
    newDenominator = self.denominator*other.denominator
    return Rational(newNumerator, newDenominator)
```

- Zauważmy, że metoda add pobiera dwa argumenty, self i other, podczas gdy we wcześniejszych przykładach metody tej użyto tylko z jednym argumentem. (Warto zatrzymać się w tym miejscu i przemyśleć to. Sytuacja jest analogiczna do tej, jaką wcześniej mieliśmy dla metody __init__).

Dodawanie liczb klasy **Rational** (4/5)

☞ Porządkując dotychczasowe ustalenia, rozważmy następujący ciąg poleceń:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1.add(r2)
```

- Konstrukcja r1.add(r2) robi coś naprawdę ważnego: wywołuje ona metodę add klasy Rational.
- Na pierwszy rzut wydaje się, że do metody add jest przekazywany tylko parametr r2, ale to złudzenie! W rzeczywistości jest przekazywane dwie wartości: po pierwsze jest automatycznie przekazywana referencja do r1 a następnie jest przekazywane r2.
- I bardzo dobrze, ponieważ z kodu metody add wynika, że są oczekiwane dwa argumenty: self i other. A zatem r1 idzie do „przegródki” self a r2 idzie do przegródki other.
- Teraz metoda add może operować na tych dwóch liczbach Rational, dodać je, utworzyć nową liczbę Rational, która jest ich sumą, a następnie zwrócić ten nowy obiekt.

☞ Zauważmy, że Python po prostu wie, że pierwszym argumentem jest zawsze przekazana automatycznie referencja do obiektu występującego przed „kropką”.

- Całe piękno, zawarte w tym, na początku dziwnie wyglądającym, sposobie działania polega na tym, że metoda zostaje użyta przez obiekt i metoda ta „wie” który obiekt jej użył.

Dodawanie liczb klasy **Rational** (5/5)

- Po wykonaniu poleceń:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1.add(r2)
```

możemy wyświetlić licznik i mianownik liczby r3 klasy Rational.

```
>>> r3.numerator
5
>>> r3.denominator
6
```

- W tym przypadku licznikiem jest 5 a mianownikiem 6.
- Warto odnotować, że zamiast polecenia r3 = r1.add(r2) użytego w omówionym przykładzie można byłoby napisać r3 = r2.add(r1). W takim przypadku r2 wywoływałby metodę add, przekazując r2 dla self a r1 dla other. Otrzymalibyśmy ten sam wynik jak poprzednio, ponieważ dodawanie liczb wymiernych jest przemienne.

Przeciążanie (1/4)

- Utworzyliśmy zatem podstawową klasę reprezentującą liczby wymierne. Jest ciekawa i przydatna, ale teraz zostanie dodatkowo uatrakcyjniona.
- Zapewne zwróciliśmy uwagę, że składnia dodawania dwóch liczb wymiernych jest trochę skomplikowana.
  - Chodzi o to, że na co dzień, dodając dwie liczby całkowite, jak 42 i 47, nie piszemy 42.add(47), lecz 42+47.
- Okazuje się, że możemy użyć operatora „+” także do dodawania liczb Rational!
  - Oto jak można tego dokonać: Po prostu zmieniamy nazwę metody add na __add__. Mamy tu dwa znaki podkreślenia przed i po słowie add. Python ma cechę, która mówi, że „jeśli pewna funkcja nosi nazwę __add__ to wtedy, jeśli użytkownik napisze r1 + r2, ja przekształcę to na r1.__add__(r2)”.
  - Skąd Python wie, że to dodawanie jest akurat dodawaniem liczb Rational a nie dodawaniem liczb całkowitych (które jest operacją wbudowaną)? On po prostu wykrywa że r1 jest klasy Rational, stąd występujący dalej znak „+” musi reprezentować metodę __add__ klasy Rational.
  - W podobny sposób możemy definiować metody __add__ dla innych klas a Python określi, której z nich należy użyć kierując się typem danej występującej przed znakiem „+”.
- Cecha ta jest nazywana przeciążaniem.

Przeciążanie (2/4)

- Przeciążyliśmy znak „+” w celu nadania mu znaczenia w zależności od kontekstu, w którym występuje. Wiele, chociaż nie wszystkie, spośród języków programowania wspiera przeciążenie.
  - W języku Python, przeciążanie dodawania stanowi zaledwie wierzchołek góry lodowej. Python umożliwia przeciążanie wszystkich standardowych operatorów arytmetycznych oraz wszystkich operatorów porównania między innymi takich jak „==”, „!=” czy „<”.
- Zastanówmy się teraz nad kwestią stwierdzania równości liczb wymiernych.
  - Rozważmy następującą sytuację, w której występują dwa różne obiekty Rational które porównujemy ze sobą:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 2)
>>> r1 == r2
False
```

- Dlaczego Python odpowiedział „False”?
  - Powodem jest to, że, mimo iż r1 i r2 wydają nam się takie same, to jednak są one referencjami (odwołaniami) do różnych obiektów. Te dwa obiekty mają taką samą zawartość, niemniej są rozróżniane, tak jak dwaj identyczni bliźniacy są różnymi ludźmi.
  - Inaczej można ten fakt postrzegać tak, że r1 i r2 są referencjami do różnych miejsc w pamięci i kiedy pojawia się pytanie czy r1 == r2 to system stwierdza, że nie.

Przeciążanie (3/4)

- Ponieważ nie powiedzieliśmy systemowi Python jak inaczej ma porównywać liczby Rational on zwyczajnie porównuje ze sobą r1 i r2 pod kątem tego, czy wskazują one dokładnie na ten sam obiekt.
  - W tej sytuacji zastosujmy „przeciążanie” znaku „==“, tak aby korespondował on z funkcją realizującą porównanie zgodnie z naszym zamysłem. Chcielibyśmy mianowicie, aby dwie liczby wymierne były uznawane za równe wtedy, gdy reprezentujące je ułamki są równe, nawet jeśli liczniki i mianowniki tych ułamków nie są takie same.
- Na przykład $\frac{1}{2} = \frac{42}{84}$.
  - Możliwym sposobem sprawdzenia, czy równość zachodzi jest znana ze szkoły metoda „mnożenia na krzyż”: Mnożymy licznik jednego z ułamków przez mianownik drugiego i sprawdzamy czy wynik jest równy iloczynowi drugiej pary licznik-mianownik.
  - Do klasy liczb wymiernych Rational dodamy metodę __eq__, której zadaniem będzie sprawdzanie równości.

```
def __eq__(self, other):
    return self.numerator * other.denominator == self.denominator * other.numerator
```

Przeciążanie (4/4)

- Teraz, mając dwie liczby wymierne r1 i r2, możemy użyć metody __eq__ pisząc r1.__eq__(r2) lub r2.__eq__(r1).
  - Jednak, skoro użyliśmy dla tej metody specjalnej nazwy __eq__ , to dla systemu Python jest to informacja, że każdy napotkany zapis r1 == r2 powinien być przekształcony na r1.__eq__(r2).
  - Istnieje wiele innych symboli operacji które można przeciążać. (Pełna lista odnośnych metod jest dostępna na stronie http://docs.python.org/2/reference/datamodel.html#special-method-names).
- Na przykład, możemy przeciążyć symbol “>=” definiując metodę o nazwie __ge__ (co znacza większe lub równe).
  - Podobnie jak __eq__, metoda ta pobiera dwa argumenty: referencję do obiektu wywołującego, która jest przekazywana automatycznie (self) oraz referencję do innego obiektu, z którym porównujemy. A zatem nasza metoda __ge__ może wyglądać tak:

```
def __ge__(self, other):
    return self.numerator * other.denominator >= self.denominator * other.numerator
```

  - Zauważmy, że różnica w implementacji metod __eq__ oraz __ge__ jest nieznaczna. Warto poświęcić chwilę na to, aby zorientować się, jak działa __ge__ .

Rozwiązanie problemu z ilością paliwa

- Wróćmy teraz do wcześniej sformułowanego problemu z paliwem.
  - Warto przypomnieć, że z uwagi na niedokładności numeryczne liczb zmiennoprzecinkowych, nasza misja (lot pojazdem kosmicznym) zakończyła się fiaskiem:

```
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```

  - Inaczej jest teraz, kiedy do uratowania misji możemy użyć nowej klasy Rational!

```
>>>  fuelNeeded = Rational(42, 1000)
>>>  tank1 = Rational(36, 1000)
>>>  tank2 = Rational(6, 1000)
>>>  tank1 + tank2 >= fuelNeeded
True
```

Misja wypełniona!

Drukowanie obiektu (1/4)

- I tak otrzymaliśmy całkiem przydatną klasę Rational.
  - Teraz jednak spójrzmy na to:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1 + r2
>>> r3
<__main__.Rational object at 0x033A8640>
>>> print(r3)
<__main__.Rational object at 0x033A8640>
```

- Uwagę zwraca tutaj dziwny efekt jaki wywołuje próba wyświetlenia r3 lub wywołania funkcji print(r3).
- W obu przypadkach Python informuje nas, że: „r3 jest obiektem Rational i nadałem mu specjalną nazwę wewnętrzną postaci 0x coś tam, coś tam…”

- A tak naprawdę przydałoby się, aby, przynajmniej w przypadku polecenia print(r3), Python wyświetlał r3 w jakiejś zgrabnej formie, która pozwalałaby odczytać wartość tej liczby.

- Przypomnijmy, że Python potrafi „przekształcać” liczby całkowite i zmienno-przecinkowe na łańcuchy znaków przy pomocy funkcji wbudowanej str. Na przykład:

```
>>> str(1)
'1'
>>> str(3.142)
'3.142'
```


Drukowanie obiektu (2/4)

- Skoro funkcja print potrafi operować na łańcuchach znaków, to możemy wyświetlać liczby następująco:

```
>>> print(str(1))
1
>>> print("My favorite number is " + str(42))
My favorite number is 42
```

- W rzeczywistości użycie funkcji str przewidziano także dla innych typów języka Python, jak listy i słowniki:

```
>>> myList = [1, 2, 3]
>>> print("Here is a very nice list: " + str(myList))
Here is a very nice list: [1, 2, 3]
```

- Okazuje się, że Python pozwala zdefiniować funkcję str dla naszej własnej klasy, poprzez przeciążenie specjalnej metody o nazwie __str__.
  - Na przykład, dla klasy Rational metodę __str__ moglibyśmy napisać następująco:

```
def __str__(self):
    return str(self.numerator) + "/" + str(self.denominator)
```

Drukowanie obiektu (3/4)

```
def __str__(self):
    return str(self.numerator) + "/" + str(self.denominator)
```

- Co zwraca ta funkcja?
  - Otrzymujemy łańcuch zawierający licznik, po którym następuje ukośnik, a po nim mianownik.
- Po napotkaniu polecenia print(str(r3)), Python zastosuje metodę __str__.
  - W trakcie wykonywania, najpierw jest wywoływana funkcja str z argumentem self.numerator. Czy wywołanie str(self.numerator) jest rekursywne? Nie! Skoro self.numerator jest liczbą całkowitą, Python wywołuje tutaj metodę str dla liczb całkowitych, po to, aby uzyskać postać znakową dla tej liczby.
  - Następnie „dokleja" na koniec tego łańcucha jednoznakowy łańcuch zawierający ukośnik „/", oznaczający kreskę ułamkową.
  - Na koniec, do istniejącego łańcucha jest doklejana postać znakowa mianownika.
  - Tak otrzymany łańcuch jest zwracany.

Drukowanie obiektu (4/4)

- W rozważanym przykładzie gdzie r3 jest liczbą wymierną $\frac{5}{6}$ możemy użyć przedstawionej na poprzednim slajdzie metody str następująco:

```
>>> r3
<__main__.Rational object at 0x0377F100>
>>> print("Here is r3: "+str(r3))
Here is r3: 5/6
```

  - Zauważmy, że w pierwszej linii, kiedy pytamy o r3, Python po prostu informuje, że jest to odnośnik (referencja) do obiektu Rational.
  - W linii trzeciej pojawia się „prośba" o zamianę r3 na łańcuch znaków tak, aby można było użyć funkcji print.
- Przy okazji warto wspomnieć, że metoda __str__ ma blisko spokrewnioną z nią metodę o nazwie __repr__ (informacji na ten temat można poszukać w Sieci).

Mutowalność (1/2)

- Załóżmy, że z jakichś powodów chcielibyśmy zmienić wartość licznika liczby r1 z aktualnej wartości na 42. Moglibyśmy po prostu napisać:

```
r1.numerator = 42
```

  - Innymi słowy, składowe obiektu Rational mogą być zmieniane. Mówimy wtedy, że klasa Rational jest mutowalna (*mutable*).
  - W języku Python, klasy, które sami definiujemy są mutowalne (chyba że umiejętnie nadamy im specjalne cechy powodujące, że będzie inaczej).

- Aby w sposób pełny docenić znaczenie mutowalności obiektów, rozważmy następującą parę funkcji:

```
def foo():
  r = Rational(1, 3)
  bar(r)
  print (r)

def bar(number):
  number.numerator += 1
```


Mutowalność (2/2)

```
def foo():
  r = Rational(1, 3)
  bar(r)
  print (r)

def bar(number):
  number.numerator += 1
```

- Co się dzieje, kiedy odwołujemy się do funkcji foo?
  - Zauważmy, że funkcja bar nic nie zwraca.
  - Jednakże przekazywana do niej zmienna number jest bezsprzecznie liczbą Rational, a foo zwiększa wartość licznika tej zmiennej.
  - Ponieważ klasy zdefiniowane przez użytkownika, takie jak Rational są mutowalne, to w przekazanym obiekcie Rational zostanie zmieniony licznik!
- Jak to naprawdę działa?
  - Zauważmy, że w funkcji foo zmienna r jest referencją do liczby $\frac{1}{3}$ klasy Rational. Innymi słowy, nasz obiekt klasy Rational znajduje się gdzieś w pamięci komputera, a r jest adresem tego miejsca. Kiedy foo wywołuje bar(r) to przesyła referencję (miejsce w pamięci) r. Teraz zmienna number odwołuje się do tego miejsca w pamięci. Kiedy Python napotyka polecenie number.numerator += 1, to najpierw udaje się do miejsca w pamięci wskazywanego przez number, następnie interpretuje „zapis kropkowy” kierujący do składowej numerator tego obiektu i zwiększa wartość tej składowej o 1. Kiedy bar w końcu zwraca sterowanie do funkcji wywołującej, foo, zmienna r w foo nadal odwołuje się do tego samego miejsca w pamięci, ale teraz licznik tam umieszczony ma nową wartość nadaną przez bar i wynoszącą 2.

Listy także są obiektami! (1/2)

- Można dalej dojść do zaskakującego odkrycia, że wszystko w języku Python jest obiektem!
- Na przykład dana typu lista jest obiektem, chociaż składnia poleceń operujących na listach w żaden sposób nie przypomina tej, jakiej używaliśmy dla obiektów Rational. Przyjrzyjmy się temu bliżej.
  - W przypadku Rational nowy obiekt był tworzony następująco:

```
r = Rational(1, 3)
```

  - Natomiast nową listę można utworzyć nieco prościej:

```
myList = [42, 1, 3]
```

  - W rzeczywistości ta notacja listowa, którą znamy i lubimy, jest po prostu udogodnieniem oferowanym użytkownikom przez projektantów Python'a. W istocie jest ona skrótem od:

```
myList = list()
myList.append(42)
myList.append(1)
myList.append(3)
```

  - Teraz, gdy poprosimy o pokazanie myList Python wyświetli [42, 1, 3].
  - Zauważmy, że polecenie myList = list() jest analogiczne do r = Rational(1, 3) z taką różnicą, że dla listy nie podajemy żadnej zawartości początkowej.
  - Następnie, append klasy lista jest stosowana do dopisywania elementów na koniec zadanej listy. Listy są mutowalne, dlatego każde takie dopisywanie zmienia listę!

Listy także są obiektami! (2/2)

- W rzeczywistości z klasą lista związanych jest wiele innych metod, o których można dowiedzieć się z sieci.
  - Na przykład, metoda reverse odwraca listę. Można to pokazać, działając na wcześniej utworzonym obiekcie myList klasy lista.
  - Warto podkreślić, że rozważana metoda reverse nie zwraca nowej listy tylko mutuje listę, dla której została wywołana.

```
>>> myList
[42, 1, 3]
>>> myList.reverse()
>>> myList
[3, 1, 42]
```

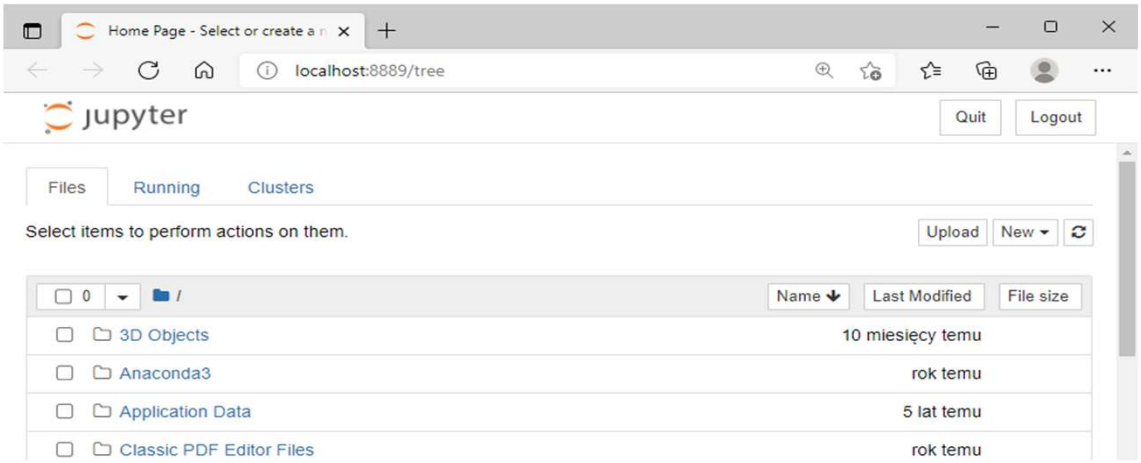
- Zatrzymajmy się na chwilę przy notacji używanej do połączenia dwóch list:

```
>>> [42, 1, 3] + [4, 5]
[42, 1, 3, 4, 5]
```

  - Dlaczego można tutaj użyć symbolu „+" ? Tak – mamy tu do czynienia z przeciążaniem w klasie lista! Inaczej mówiąc, jest to metoda __add__ w tej klasie!
- Łańcuchy znaków, słowniki, a nawet liczby całkowite i zmiennoprzecinkowe są w języku Python obiektami!
  - Jednakże, wiele spośród typów wbudowanych – jak: łańcuchy znaków, liczby całkowite czy liczby zmiennoprzecinkowe zostały zaprojektowane jako niemutowalne.
  - Oznacza to, że ich wewnętrzna struktura nie może być zmieniana. Można także definiować własne obiekty niemutowalne, ale to wymagałoby dodatkowych działań i rzadko kiedy jest konieczne.

Podstawy użycia grafiki 3D (1/2)

- Aby rozpocząć pracę z grafiką 3D (trójwymiarową) należy zainstalować oprogramowanie Anaconda3 oraz uruchomić Anaconda Navigator.
- Dalej wybieramy jupyter notebook, co powoduje otwarcie w przeglądarce następującego okna:



Podstawy użycia grafiki 3D (2/2)

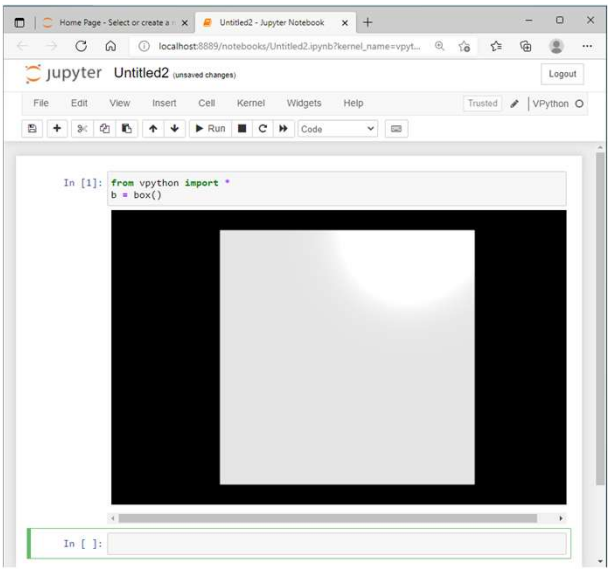
- W tym oknie należy użyć przycisku New umieszczonego na górze po prawej stronie ekranu, a następnie w menu Notebooks wybrać VPython.
  - Otwiera się nowe okno, które pozwala zaimportować vpython poleceniem:

    ```
    from vpython import *
    ```

  - Następnie, w tej samej komórce (*cell*) wprowadzamy:

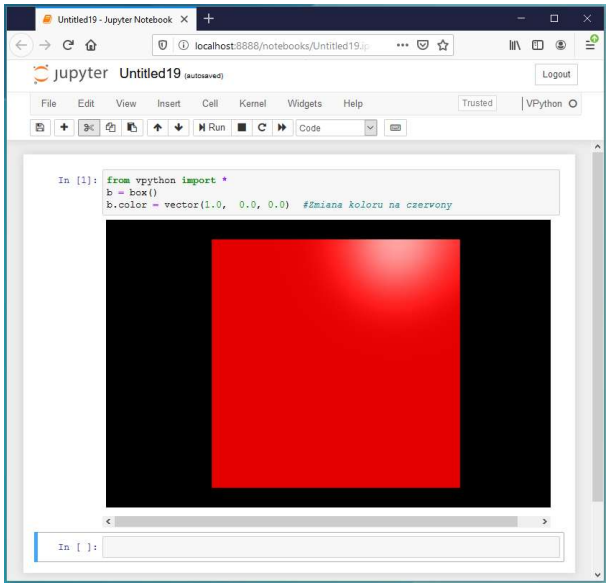
    ```
    b = box()
    ```

  - Na koniec wciskamy Run. Poniżej komórki (tj. pola gdzie wpisywaliśmy polecenia), powinno pojawić się białe pudełko.



Modyfikowanie kodu

- Jeśli zachodzi potrzeba zmodyfikowania kodu oraz jego ponownego uruchomienia, należy użyć przycisku Restart the Kernel (umieszczonego obok Run). W efekcie pojawi się „wyskakujące okienko”, w którym wciskamy czerwony przycisk Restart.
  - Następnie należy zaczekać, aż w prawym górnym rogu błyśnie niebieski komunikat Kernel Ready a potem wcisnąć Run.
  - Wynik uruchomienia zmodyfikowanego kodu, który zmienia kolor pudełka na czerwony można zobaczyć w oknie graficznym.



Klasa **box**

- Powróćmy teraz do białego pudełka uzyskanego w wyniku wykonania polecenia b=box().
  - Pudełko to może wydać się białym kwadratem, dlatego (przytrzymując prawy klawisz myszy) warto je obrócić i w ten sposób przekonać się, że jest to obiekt 3D.
  - Jak nietrudno zgadnąć, box jest klasą zdefiniowaną w module vpython. Polecenie:

```
b = box()
```

uruchamia konstruktor w celu utworzenia nowego obiektu klasy box. Nazwą tego obiektu, a bardziej precyzyjnie, referencją (odnośnikiem) do tego obiektu, staje się b.

- W analogiczny sposób jak klasa liczb wymiernych Rational miała atrybuty numerator oraz denominator, tak klasa box także ma pewną liczbę atrybutów.
  - Należą do nich takie właściwości pudełka jak: długość, wysokość i szerokość; jego położenie, kolor, a nawet cechy użytego materiału.
  - Przykład obok pokazuje, jak można zmieniać te atrybuty używając stosownych poleceń.

```
b.length = 0.5    # the box's length just changed
b.width = 2.0    # the box's width just changed
b.height = 1.5    # the box's height just changed
b.color = vector(1.0, 0.0, 0.0)    # the box turned red
b.texture = textures.wood    # it's wood-grained!
b.opacity = 0.42    # it's translucent!
```

Atrybuty pudełka

- W chwili utworzenia pudełka poleceniem b = box() jego atrybutom są nadawane wartości „domyślne”.
  - Atrybutom długość, szerokość oraz wysokość nadano jednakową wartość równą 1.0, kolor przyjęto jako biały, a tekstura pudełka miała podstawowy, monotonny charakter.
- Zauważmy, że niektóre atrybuty pudełka są dosyć oczywiste: długość, szerokość oraz wysokość są wartościami liczbowymi.
- Jednak atrybut opisujący kolor już zwraca uwagę. Także charakterystyka odnosząca się do tekstury została ustawiona w nieco dziwny sposób.
  - Przyjrzyjmy się trochę bliżej tylko jednemu z tych atrybutów, a o innych można dowiedzieć się nieco więcej ze strony internetowej z dokumentacją VPython.
  - Warto sięgnąć po „dokumentację Glowscript”. Jeśli szukać tylko „dokumentacji vpython” trafimy na opis „Classic VPython”, który był poprzednikiem Jupyter Vpython.

Atrybut **color**

- Wyczyścimy teraz bieżącą komórkę (zostawiamy tylko polecenie import).
- Dalej tworzymy nowe pudełko i wprowadzamy zapytanie o jego kolor używając polecenia:

```
c = box()
c.color
<1, 1, 1>
```

- W środowisku VPython kolory są przedstawiane w formie krotki złożonej z trzech liczb z przedziału od 0.0 do 1.0.
  - Trzy składowe tworzące krotkę pokazują jaki jest udział odpowiednio koloru czerwonego (od wartości minimalnej 0.0 do wartości maksymalnej 1.0), zielonego oraz niebieskiego w kolorze opisywanego obiektu.
  - A zatem, krotka (1.0,1.0,1.0) oznacza, że chcemy mieć maksimum każdego z kolorów, co w efekcie daje kolor biały. Krotka (1.0,0.0,0.0) to kolor jasnoczerwony a krotka (0.7,0.0,0.4) to mieszanka sporej ilości czerwonego z nieco mniejszą ilością niebieskiego.

Atrybut **pos**

- Klasa box ma także inny atrybut o nazwie pos który przechowuje położenie środka pudełka.
- Układ współrzędnych wykorzystywany w VPython jest określony przez tzw. „zasadę prawej ręki”:
  - Jeśli w prawej ręce wyprostujemy kciuk, palec wskazujący oraz palec środkowy tak, aby były względem siebie prostopadłe, a rękę ułożymy dłonią do siebie, wtedy kciuk wskazuje dodatni kierunek osi x, palec wskazujący wyznacza dodatni kierunek osi y, a palec środkowy wskazuje dodatni kierunek osi z.
- Inaczej mówiąc, zanim zaczniemy obracanie przy pomocy myszy, osią poziomą jest oś x, osią pionową - oś y, a oś z pokazuje kierunek „od ekranu”.
- Określmy pozycję pudełka dodając polecenie c.pos. Wykonując nowy kod, otrzymamy:

```
c.pos
<0, 0, 0>
```


Użycie klasy **vector**

- W systemie VPython używa się klasy vector, a pos jest obiektem tego typu, na co wskazuje użyta notacja wektorowa.
  - Sprytne! Klasa box została zdefiniowana przy użyciu klasy vector.
- Użycie klasy vector w tamach klasy box jest bardzo eleganckim rozwiązaniem!
  - No dobrze, ale skąd akurat wektor? Dlaczego nie moglibyśmy zamiast niego użyć krotki albo listy?
- Oto w czym rzecz: Wewnątrz klasy vector zdefiniowano pewne metody wykonujące operacje na wektorach.
  - Na przykład, klasa vector ma przeciążony operator dodawania dwóch wektorów:

```
v = vector(1, 2, 3)
w = vector(10, 20, 30)
v + w
<11, 22, 33>
```

  - Klasa ta ma także wiele innych operacji wektorowych. Na przykład, metoda norm() zwraca wektor, który wskazuje ten sam kierunek ale jego moduł (długość) wynosi 1:

```
u = vector(1, 1, 0)
u.norm()
<0.707107, 0.707107, 0>
```

Zmiana położenia pudełka

- A zatem gdybyśmy przedstawiali wektory za pomocą list, tracilibyśmy możliwość ich wygodnego dodawania i normalizowania, jak również wykonywania innych ważnych operacji na wektorach.
- Jednak naszym obecnym celem jest modyfikacja wektora pos dla naszego pudełka , co pozwoli na jego przemieszczanie.
  - Możemy tego dokonać, na przykład, tak:

```
c.pos = vector(0, 1, 2)
```

- Chociaż zawsze można najpierw utworzyć pudełko, a dopiero potem zmieniać jego atrybuty, czasem wygodniej jest po prostu ustawić wartości atrybutów w chwili inicjalizacji pudełka.
  - Konstruktor klasy box pozwala nadać wartości atrybutom w chwili konstruowania pudełka następująco:

```
d = box(length = 0.5, width = 2.0, height = 1.5, color = vector(1.0, 0.0, 0.0))
```

- Jeśli którykolwiek z atrybutów zostanie pominięty, będzie mu nadana wartość domyślna.
  - Na przykład, skoro nie podaliśmy wartości dla wektora pos, to pierwotnym położeniem pudełka będzie początek układu współrzędnych.

Obracanie form przestrzennych

- Poza pudelkami VPython udostępnia szereg innych klas odpowiadających formom przestrzennym, wśród których są m.in.: kule, stożki i cylindry.
- Chociaż obiekty te mają pewne specyficzne dla siebie atrybuty (na przykład kula ma promień), to wszystkie one korzystają z jednakowych przydatnych metod.
- Jedną z tych metod jest rotate (obracanie).
  - Nie jest niespodzianką, że metoda ta umożliwia obracanie odnośnego obiektu.
  - Spróbujmy wykonać pojedynczy obrót używając rotate!
- Dokonamy tego dla wcześniej utworzonego pudełka d:

```
d.rotate(angle=pi/4)
```

  - W tak sformułowanym poleceniu zadysponowano obrót pudełka d o $\frac{\pi}{4}$ radianów.
  - Obrót ten jest domyślnie określony w radianach wokół osi x.

Obracanie czerwonego pudełka

- Teraz spróbujmy zebrać to, co już wiemy i napisać dwa proste programy w VPython pokazując swoje obecne możliwości w zakresie grafiki 3D.
- Na początek będzie to bardzo krótki program który powoduje ciągłe obracanie się czerwonego pudełka (przed skopiowaniem proponowanego kodu należy wyczyścić odnośną komórkę).

```
from vpython import *

def spinBox():
   myBox = box(color = vector(1.0, 0.0, 0.0))
   while True:
      # Slow down the animation to 60 frames per second.
      # Change the value to see the effect!
      rate(60)
      myBox.rotate(angle=pi/100)
spinBox()
```


Jeszcze jeden prosty program

- Rozważymy drugi przykład. Przed uruchomieniem warto przeanalizować zawarte w nim polecenia.

```
from vpython import *
import random

def spinboxes():
   boxList = []
   for boxNumber in range(0,10):
      x = random.randint(-5,5)
      y = random.randint(-5,5)
      z = random.randint(-5,5)
      red = random.random()      # random number between 0 and 1
      green = random.random()    # random number between 0 and 1
      blue = random.random()     # random number between 0 and 1
      newBox = box()
      newBox.pos = vector(x, y, z)
      newBox.color = vector(red, green, blue)
      # makes boxes rotate in random directions
      newBox.axis = random.choice([vector(1,0,0),vector(0,1,0),vector(0,0,1)])
      boxList.append(newBox)
   # the physics loop, which updates the world
   while True:
      rate(60)
      for myBox in boxList:
         myBox.rotate(angle=pi/100)
spinboxes()
```