Rozdział 14

Programowanie obiektowe

14.1. Zarządzanie większymi programami

Na początku książki omówiliśmy cztery podstawowe wzorce programowania, których używamy do tworzenia programów:

- Kod sekwencyjny
- Kod warunkowy (instrukcje `if`)
- Kod powtarzalny (pętle)
- Zapisanie i ponowne użycie (funkcje)

W późniejszych rozdziałach analizowaliśmy proste zmienne, jak również struktury danych, takie jak listy, krotki i słowniki.

Podczas tworzenia programów projektujemy struktury danych i piszemy kod służący do operowania tymi strukturami danych. Istnieje wiele sposobów pisania programów i prawdopodobnie do tej pory napisałeś już kilka programów, które nie są „zbyt eleganckie", oraz kilka innych programów, które są „bardziej eleganckie". Mimo że Twoje programy mogą być małe, to zapewne zaczynasz dostrzegać, że pisanie kodu wymaga odrobiny artyzmu i estetyki.

W miarę jak programy rozrastają się do milionów wierszy, coraz ważniejsze staje się pisanie kodu, który jest łatwy do zrozumienia. Jeśli pracujesz nad programem o długości miliona linii, nigdy nie dasz rady mieć w głowie całego programu naraz. Potrzebujemy sposobów, aby rozbić duże programy na wiele mniejszych kawałków, tak abyśmy mieli mniej kodu do przeglądania, rozwiązując jakiś problem, naprawiając błędy lub dodając nowe funkcje.

W pewnym sensie programowanie obiektowe jest sposobem na uporządkowanie kodu tak, abyś mógł skupić się na 50 liniach i go zrozumieć, ignorując na chwilę pozostałe 999 950 linii kodu.

14.2. Rozpoczęcie pracy

Podobnie jak w przypadku wielu innych aspektów programowania, poznanie koncepcji programowania obiektowego jest konieczne, zanim będzie można je skutecznie wykorzystać. Powinieneś potraktować ten rozdział jako sposób na poznanie niektórych terminów i pojęć oraz popracować z kilkoma prostymi przykładami, aby stworzyć podstawy do dalszej nauki.

Przeczytanie tego rozdziału powinno przede wszystkim dać Ci podstawowe zrozumienie, jak konstruowane są obiekty i jak one funkcjonują, a co najważniejsze, jak wykorzystujemy możliwości obiektów, które są nam dostarczane przez Pythona i jego biblioteki.

14.3. Korzystanie z obiektów

Jak się okazuje, w tej książce przez cały czas używaliśmy obiektów. Python dostarcza nam wiele wbudowanych obiektów. Oto prosty kod, w którym kilka pierwszych linii powinno być już dla Ciebie bardzo naturalnych i zrozumiałych.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Kod źródłowy: https://py4e.pl/code3/party1.py
```

Zamiast skupiać się na tym, ostatecznym wyniku działania tych linii, spójrzmy na to, co *naprawdę* dzieje się w tym kodzie z punktu widzenia programowania obiektowego. Nie martw się, jeśli poniższe akapity nie będą miały dla Ciebie żadnego sensu po pierwszym przeczytaniu – nie zdefiniowaliśmy jeszcze wszystkich użytych tam terminów.

Pierwsza linia *konstruuje* obiekt typu `list`, druga i trzecia *wywołuje metodę* `append()`, czwarta linia wywołuje metodę `sort()`, a piąta linia *pobiera* (*wyszukuje*) element z pozycji 0.

Szósta linia wywołuje metodę `__getitem__()` na liście `stuff` z parametrem zero.

```
print (stuff.__getitem__(0))
```

Siódma linia to jeszcze dłuższy zapis odzyskania elementu z zerowej pozycji listy `stuff`.

```
print (list.__getitem__(stuff,0))
```

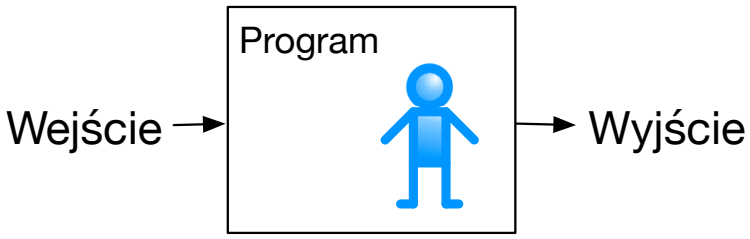
W powyższym fragmencie kodu wywołujemy metodę `__getitem__` w klasie `list` i *przekazujemy* jako parametry listę oraz pozycję elementu, który chcemy pobrać z tej listy.

Ostatnie trzy wiersze programu są równoważne, ale wygodniej jest po prostu użyć składni z nawiasami kwadratowymi, żeby wyszukać element znajdujący się na konkretnym miejscu listy.

Możemy przyjrzeć się możliwościom danego obiektu, patrząc na wynik funkcji `dir()`:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Pozostała część tego rozdziału zdefiniuje wszystkie powyższe terminy, więc pamiętaj, by po jego zakończeniu wrócić do tej sekcji i ponownie przeczytać powyższe akapity i sprawdzić czy je rozumiesz.



Rysunek 14.1. Program

14.4. Zaczynając od programów...

Program w swojej najbardziej podstawowej formie pobiera pewną ilość danych wejściowych, przetwarza je i wytwarza pewną ilość danych wyjściowych. Nasz program do konwersji numerów pięter to bardzo krótki, ale kompletny przykład pokazujący te trzy kroki.

```
usf = input('Wprowadź numer piętra w zapisie amerykańskim: ')
wf = int(usf) - 1
print('Numer piętra w zapisie nieamerykańskim to', wf)

# Kod źródłowy: https://py4e.pl/code3/elev.py
```

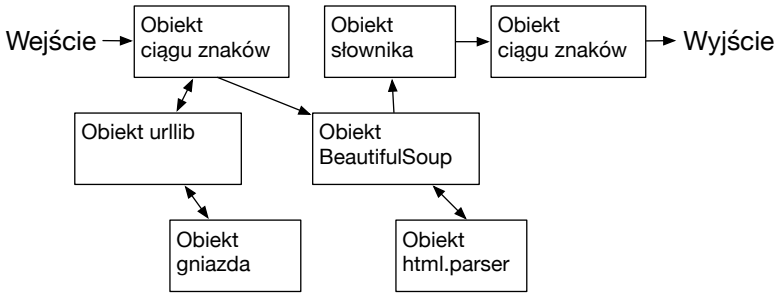
Jeśli zastanowimy się trochę dłużej nad tym programem, to zauważymy, że istnieje „świat zewnętrzny" oraz nasz program. Wejście i wyjście są tymi miejscami, w których program wchodzi w interakcję ze światem zewnętrznym. Wewnątrz programu mamy kod i dane do wykonania zadania, do którego jest przeznaczony ten program.

Jednym ze sposobów na myślenie o programowaniu obiektowym jest rozdzielenie naszego programu na wiele „stref". Każda strefa zawiera pewien kod i dane (tak jak program) oraz ma dobrze zdefiniowane interakcje ze światem zewnętrznym i innymi strefami w programie.

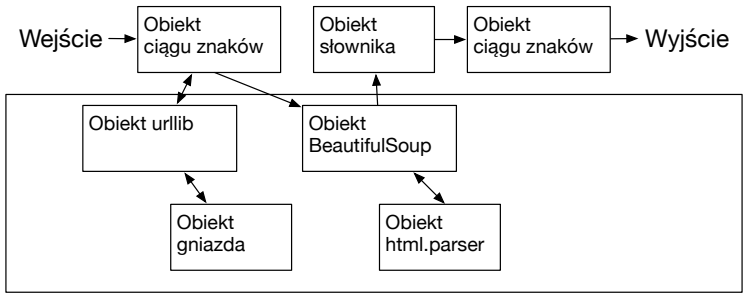
Jeśli spojrzymy ponownie na aplikację do wyodrębniania linków, w której korzystaliśmy z biblioteki BeautifulSoup, to możemy zobaczyć program, który jest skonstruowany poprzez złączenie różnych obiektów w celu wykonania tego zadania:

```
import urllib.request
from bs4 import BeautifulSoup
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Kod źródłowy: https://py4e.pl/code3/urllinks.py
```

Wczytujemy adres URL do zmiennej przechowującej ciąg znaków, a następnie przekazujemy ją do `urllib`, tak aby pobrać dane z sieci. Moduł `urllib` wykorzystuje w rzeczywistości moduł `socket` do nawiązania



Rysunek 14.2. Program jako sieć obiektów



Rysunek 14.3. Pomijanie szczegółów podczas używania obiektu

połączenia z siecią w celu pobrania danych. Bierzemy ciąg znaków, który zwraca `urllib`, i przekazujemy go do `BeautifulSoup` do dalszej analizy. `BeautifulSoup` korzysta z obiektu `html.parser`[1] i w wyniku też zwraca nam obiekt. Na zwróconym obiekcie wywołujemy metodę `tags()`, która zwraca słownik obiektów będących znacznikami. Przechodzimy w pętli po znacznikach i dla każdego znacznika wywołujemy metodę `get()`, tak aby wypisać jego atrybut `href`.

Możemy narysować dla tego programu diagram z oznaczeniem, jak te obiekty ze sobą współpracują.

W tym przypadku kluczowe nie jest idealne zrozumienie jak ten program działa, ale zobaczenie, jak w celu stworzenia programu budujemy sieć współdziałających ze sobą obiektów i zarządzamy przepływem informacji pomiędzy nimi. Należy również zauważyć, że gdy kilka rozdziałów temu natrafiłeś na ten program, to mogłeś w pełni zrozumieć, co się w nim dzieje, nawet nie zdając sobie sprawy z tego, że program „zarządzał przepływem danych pomiędzy obiektami". To były po prostu tylko linie kodu, które wykonały swoje zadanie.

14.5. Dzielenie problemu na mniejsze podproblemy

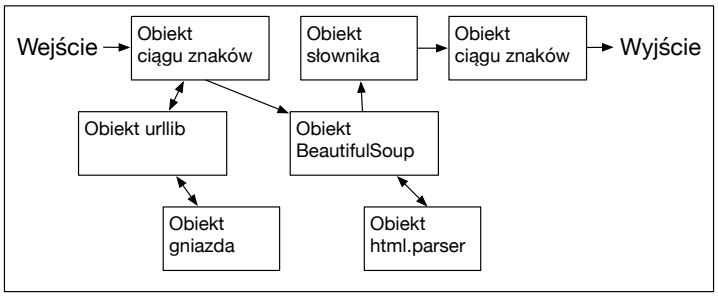
Jedną z zalet podejścia obiektowego jest to, że może ono ukryć złożoność jakiegoś problemu. Na przykład, o ile musimy wiedzieć, jak używać biblioteki `urllib` i `BeautifulSoup`, to nie musimy wiedzieć, jak one działają w środku. Pozwala nam to skupić się na tej części problemu, którą musimy rozwiązać, i pominąć pozostałe części programu.

Możliwość skupienia się wyłącznie na tej części programu, na której nam zależy, i pominięcia reszty jest również pomocna dla twórców używanych przez nas obiektów. Na przykład programiści tworzący `BeautifulSoup` nie muszą wiedzieć ani dbać o to, w jaki sposób pobieramy naszą stronę HTML, jakie części chcemy przeczytać lub co planujemy zrobić z danymi, które pobieramy ze strony.

14.6. Nasz pierwszy obiekt w Pythonie

W podstawowym zakresie obiekt to po prostu jakiś kod plus struktury danych, które są mniejsze niż cały program. Zdefiniowanie funkcji pozwala nam na zapisanie krótkiego kodu i nadanie mu nazwy, a następnie wywołanie tego kodu za pomocą nazwy funkcji.

[1] https://docs.python.org/3/library/html.parser.html



Rysunek 14.4. Pomijanie szczegółów podczas budowania obiektu

Obiekt może zawierać szereg funkcji (które nazywamy *metodami*), jak również dane, które są wykorzystywane przez te funkcje. Dane będące częścią obiektu nazywamy *atrybutami*.

Zawsze gdy tworzymy obiekt, używamy słowa kluczowego `class` do zdefiniowania danych i kodu, które będą się na niego składały. Słowo kluczowe `class` zawiera nazwę klasy i rozpoczyna wcięty blok kodu, w którym umieszczamy atrybuty (dane) i metody (kod).

```
class PartyAnimal:
    x = 0

    def party(self) :
        self.x = self.x + 1
        print("Jak na razie", self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Kod źródłowy: https://py4e.pl/code3/party2.py
```

Każda metoda wygląda jak funkcja. Zaczyna się od słowa kluczowego `def` i składa się z wciętego bloku kodu. Nasz obiekt ma jeden atrybut (`x`) i jedną metodę (`party()`). Metody mają specjalny pierwszy parametr, który nazywamy umownie `self`.

Tak jak słowo kluczowe `def` nie powoduje wykonania kodu funkcji, tak samo słowo kluczowe `class` nie tworzy obiektu. Zamiast tego definiuje ono szablon mówiący o tym, jakie dane i kod będą zawarte w każdym obiekcie typu `PartyAnimal`. Klasa jest jak foremka do wykrawania ciastek, a obiekty tworzone przy jej użyciu to ciasteczka[2]. Nie umieszczasz lukru na foremce do wykrawania ciastek – lukier umieszczasz na ciasteczkach. A na każdym ciasteczku możesz umieścić inny lukier!

Kontynuując analizę naszego przykładowego programu, dochodzimy do pierwszej wykonywalnej linii kodu:

```
an = PartyAnimal()
```

Tutaj instruujemy Pythona, by skonstruował (tzn. stworzył) *obiekt* lub *instancję* klasy `PartyAnimal`. Wygląda to jak wywołanie funkcji o nazwie klasy. Python konstruuje obiekt z odpowiednimi danymi i metodami i zwraca obiekt, który jest następnie przypisany do zmiennej `an`. W pewnym sensie jest to dość podobne do poniższej linii, której używaliśmy często już wcześniej:

```
counts = dict()
```

[2]Prawa autorskie do obrazu ciasteczka: CC-BY https://www.flickr.com/photos/dinnerseries/23570475099

Rysunek 14.5. Klasa i dwa obiekty

Tutaj instruujemy Pythona, by skonstruował obiekt przy użyciu szablonu `dict` (obecnego już w Pythonie), zwrócił instancję słownika i przypisał ją do zmiennej `counts`.

Klasa `PartyAnimal` jest używana do konstruowania obiektu, natomiast zmienna `an` jest używana do wskazywania tego obiektu. Używamy `an`, żeby mieć dostęp do kodu i danych dla tej konkretnej instancji klasy `PartyAnimal`.

Każdy obiekt/instancja `PartyAnimal` zawiera w sobie zmienną `x` oraz metodę/funkcję o nazwie `party()`. W tej linii wywołujemy metodę `party()`:

```
an.party()
```

Kiedy metoda `party()` jest wywoływana, pierwszy parametr (który nazywamy umownie `self`) wskazuje konkretną instancję obiektu `PartyAnimal`, z której wywoływana jest metoda `party()`. W obrębie metody `party()` widzimy linię:

```
self.x = self.x + 1
```

Składnia ta, używająca operatora *kropki*, mówi „`x` wewnątrz `self`". Przy każdym wywołaniu `party()` wewnętrzna wartość `x` jest zwiększana o 1 (a później wartość ta jest wypisywana na ekran przy pomocy `print()`).

Poniższa linia przedstawia inny sposób wywołania metody `party()` wewnątrz obiektu `an`:

```
PartyAnimal.party(an)
```

W tym wariancie uzyskujemy dostęp do kodu bezpośrednio poprzez klasę i jawnie przekazujemy wskaźnik obiektu `an` jako pierwszy parametr metody (czyli `self`). Możesz myśleć o `an.party()` jako o skróconym zapisie powyższej linii.

Po uruchomieniu programu uzyskujemy następujący wynik:

```
Jak na razie 1
Jak na razie 2
Jak na razie 3
Jak na razie 4
```

Konstruujemy obiekt i czterokrotnie wywołujemy metodę `party()` zarówno zwiększając, jak i wypisując wartość `x` będącą wewnątrz obiektu `an`.

14.7. Klasy i typy

Jak już widzieliśmy, w Pythonie wszystkie zmienne mają określony typ. Możemy użyć wbudowanej funkcji `dir()` do zbadania możliwości danej zmiennej. Możemy również użyć `type()` i `dir()` z tworzonymi klasami.

```
class PartyAnimal:
    x = 0

    def party(self) :
        self.x = self.x + 1
        print("Jak na razie", self.x)

an = PartyAnimal()
print("Type", type(an))
print("Dir ", dir(an))
print("Type", type(an.x))
print("Type", type(an.party))

# Kod źródłowy: https://py4e.pl/code3/party3.py
```

Po uruchomieniu programu zobaczymy następujący wynik:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

Możesz zauważyć, że przy użyciu słowa kluczowego `class` stworzyliśmy nowy typ. Używając `dir()` możesz zobaczyć, że w obiekcie jest dostępny zarówno atrybut całkowitoliczbowy `x`, jak i metoda `party()`.

14.8. Cykl życia obiektu

W poprzednich przykładach definiowaliśmy klasę (szablon), używaliśmy jej do stworzenia instancji (obiektu) tej klasy, a następnie korzystaliśmy z instancji. Kiedy program zakończy pracę, wszystkie zmienne są porzucane. Zazwyczaj nie zastanawiamy się zbytnio nad tworzeniem i usuwaniem zmiennych. Jednak gdy nasze obiekty stają się coraz bardziej złożone, często musimy podjąć wewnątrz obiektu pewne działania, tak aby ustawić niektóre rzeczy w momencie, gdy obiekt jest konstruowany, i ewentualnie usunąć pewne rzeczy, gdy obiekt jest porzucany.

Jeśli chcemy, by nasz obiekt był świadomy momentów konstruowania i niszczenia, dodajemy do niego specjalnie nazwane metody:

```
class PartyAnimal:
    x = 0

    def __init__(self):
        print('Jestem tworzony')

    def party(self) :
        self.x = self.x + 1
        print('Jak na razie', self.x)

    def __del__(self):
```

```
        print('Jestem niszczony', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an zawiera', an)

# Kod źródłowy: https://py4e.pl/code3/party4.py
```

Po uruchomieniu programu zobaczymy następujący wynik:

```
Jestem tworzony
Jak na razie 1
Jak na razie 2
Jestem niszczony 2
an zawiera 42
```

Gdy Python tworzy nasz obiekt, to wywołuje metodę `__init__()`, tak aby dać nam szansę na ustawienie pewnych domyślnych lub początkowych wartości dla tego obiektu. Kiedy Python natrafi na linię:

```
an = 42
```

to w rzeczywistości „odrzuca nasz obiekt", żeby móc ponownie użyć zmiennej `an` do przechowywania wartości 42. Właśnie w tym momencie, w którym nasz obiekt `an` jest „niszczony", wywoływany jest nasz kod destruktora (`__del__()`). Nie możemy powstrzymać zniszczenia naszej zmiennej, ale możemy dokonać niezbędnego czyszczenia, tuż zanim nasz obiekt zostanie zniszczony.

Często podczas prac nad kodem obiektu zapada decyzja, by dodać do niego konstruktor, tak aby ustawić jego początkowe wartości. Natomiast stosunkowo rzadko się zdarzy, byśmy potrzebowali destruktora dla danego obiektu.

14.9. Wiele instancji

Jak do tej pory zdefiniowaliśmy klasę, stworzyliśmy pojedynczy obiekt, użyliśmy go, a następnie porzuciliśmy. Jednak prawdziwa moc programowania obiektowego ujawnia przy tworzeniu wielu instancji naszej klasy.

Gdy tworzymy wiele obiektów na podstawie jednej klasy, możemy potrzebować dla każdego z nich ustawić różne wartości początkowe. Dane te możemy przekazać do konstruktorów, tak aby nadać każdemu z obiektów inną wartość początkową:

```
class PartyAnimal:
    x = 0
    name = ''
    def __init__(self, nam):
        self.name = nam
        print(self.name, '- utworzenie')

    def party(self) :
        self.x = self.x + 1
        print(self.name, '- zliczenie imprezek -', self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
```

```
j.party()
s.party()

# Kod źródłowy: https://py4e.pl/code3/party5.py
```

Konstruktor posiada zarówno parametr `self`, który wskazuje na instancję obiektu, jak i dodatkowe parametry, które są przekazywane do niego w trakcie tworzenia obiektu:

```
s = PartyAnimal('Sally')
```

Wewnątrz konstruktora druga linia kopiuje parametr (`nam`), który jest przekazywany do atrybutu `name` będącego już w instancji obiektu.

```
self.name = nam
```

Wynik programu pokazuje, że każdy z obiektów (`s` i `j`) zawiera swoje własne, niezależne kopie `x` oraz `nam`:

```
Sally - utworzenie
Jim - utworzenie
Sally - zliczenie imprezek - 1
Jim - zliczenie imprezek - 1
Sally - zliczenie imprezek - 2
```

14.10. Dziedziczenie

Inną przydatną cechą programowania obiektowego jest możliwość stworzenia nowej klasy poprzez rozszerzenie już istniejącej. Podczas tej operacji klasę źródłową nazywamy *klasą bazową*, a nową klasę nazywamy *klasą pochodną* lub *klasą potomną*.

Kontynuując poprzedni przykład, przeniesiemy naszą klasę `PartyAnimal` do osobnego pliku `party.py`.

```
class PartyAnimal:
    x = 0
    name = ''
    def __init__(self, nam):
        self.name = nam
        print(self.name, '- utworzenie')

    def party(self) :
        self.x = self.x + 1
        print(self.name, '- zliczenie imprezek -', self.x)

# Kod źródłowy: https://py4e.pl/code3/party.py
```

Następnie w nowym pliku możemy „zaimportować” klasę `PartyAnimal` i rozszerzyć ją, tak jak pokazano poniżej:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
    points = 0
    def six(self):
        self.points = self.points + 6
        self.party()
        print(self.name, "- punkty -", self.points)
```

```
s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Kod źródłowy: https://py4e.pl/code3/party6.py
```

Kiedy definiujemy klasę `CricketFan`, wskazujemy, że rozszerzamy klasę `PartyAnimal`. Oznacza to, że wszystkie zmienne (`x`) i metody (`party()`) z klasy `PartyAnimal` są *dziedziczone* przez klasę `CricketFan`. Na przykład w ramach metody `six()` klasy `CricketFan` wywołujemy metodę `party()` z klasy `PartyAnimal`.

Podczas uruchomienia programu tworzymy zmienne `s` i `j` jako niezależne instancje `PartyAnimal` i `CricketFan`. Obiekt `j` ma dodatkowe możliwości w porównaniu do obiektu `s`.

```
Sally - utworzenie
Sally - zliczenie imprezek - 1
Jim - utworzenie
Jim - zliczenie imprezek - 1
Jim - zliczenie imprezek - 2
Jim - punkty - 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

W wyniku funkcji `dir()` na obiekcie `j` (instancja klasy `CricketFan`) widzimy, że obiekt ten ma atrybuty i metody klasy nadrzędnej, a ponadto atrybuty i metody, które zostały dodane poprzez rozszerzoną klasę `CricketFan`.

14.11. Podsumowanie

Powyższy rozdział jest bardzo szybkim wprowadzeniem do programowania obiektowego, które skupia się głównie na terminologii i składni używanej podczas definiowania i używania obiektów. Przejrzyjmy szybko kod, który widzieliśmy na początku rozdziału. W tym momencie powinieneś w pełni rozumieć, co się w nim dzieje.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Kod źródłowy: https://py4e.pl/code3/party1.py
```

Pierwsza linia tworzy *obiekt* typu `list`. Gdy Python tworzy obiekt typu `list`, to wywołuje metodę *konstruktora* (o nazwie `__init__()`), tak by ustawić wewnętrzne atrybuty, które będą używane do przechowywania danych listy. Nie przekazaliśmy żadnych parametrów do *konstruktora*. Gdy konstruktor zakończy działanie, używamy zmiennej `stuff`, aby móc wskazywać na zwróconą instancję klasy `list`.

Druga i trzecia linia wywołują metodę `append()` z jednym parametrem, żeby dodać nową pozycję na końcu listy, aktualizując atrybuty w ramach obiektu `stuff`. Następnie w czwartej linii wywołujemy metodę `sort()` bez parametrów, by posortować dane wewnątrz obiektu `stuff`.

W kolejnym kroku wypisujemy pierwszą pozycję z listy za pomocą nawiasów kwadratowych, które są skrótem do wywołania metody `__getitem__()` w ramach `stuff`. Jest to równoważne z wywołaniem metody

`__getitem__()` w ramach *klasy* `list` oraz przekazaniem obiektu `stuff` jako pierwszego, a szukanej pozycji jako drugiego parametru.

Na końcu programu obiekt `stuff` jest porzucany, ale nie przed wywołaniem *destruktora* (o nazwie `__del__()`), żeby obiekt, o ile jest to konieczne, mógł pod koniec swego istnienia wyczyścić wszystkie niepotrzebne już elementy.

Omówiliśmy tutaj podstawy programowania obiektowego. Jest wiele innych dodatkowych tematów, np. jak najlepiej używać podejścia obiektowego podczas tworzenia dużych aplikacji i bibliotek, jednak wykraczają one poza temat tego rozdziału.[3].

14.12. Słowniczek

atrybut Zmienna, która jest częścią klasy.

destruktor Opcjonalna, specjalnie nazwana metoda (`__del__()`), która jest wywoływana w momencie, gdy obiekt jest niszczony. Destruktory są rzadko używane.

dziedziczenie Tworzenie nowej klasy poprzez rozszerzenie klasy już istniejącej. Klasa pochodna oprócz dodatkowych atrybutów i metod zdefiniowanych w niej posiada również wszystkie atrybuty i metody klasy bazowej.

klasa Szablon, który może być użyty do stworzenia obiektu. Definiuje atrybuty i metody, które będą składały się na obiekt.

klasa bazowa Klasa, która jest rozszerzana, by utworzyć nową klasę pochodną. Klasa bazowa udostępnia klasie pochodnej wszystkie swoje metody i atrybuty.

klasa pochodna Nowa klasa utworzona w momencie rozszerzenia klasy bazowej. Klasa pochodna dziedziczy wszystkie atrybuty i metody klasy bazowej.

konstruktor Opcjonalna, specjalnie nazwana metoda (`__init__()`), która jest wywoływana w momencie, gdy klasa jest używana do tworzenia obiektu. Zazwyczaj jest ona używana do ustawienia początkowych wartości obiektu.

metoda Funkcja zawarta w klasie i obiektach, które są z niej utworzone.

obiekt Utworzona instancja klasy. Obiekt zawiera wszystkie atrybuty i metody, które zostały zdefiniowane przez klasę. Niektóre dokumentacje obiektowe używają terminu „instancja" zamiennie z terminem „obiekt".

[3] Jeśli jesteś ciekaw, gdzie jest zdefiniowana klasa `list`, to zajrzyj pod adres (miejmy nadzieję, że adres URL się nie zmieni) https://github.com/python/cpython/blob/master/Objects/listobject.c – klasa listy jest napisana w języku „C". Jeśli przejrzysz wspomniany kod źródłowy i okaże się on dla Ciebie interesujący, to być może powinieneś zrobić dodatkowo kilka kursów z dziedziny informatyki.