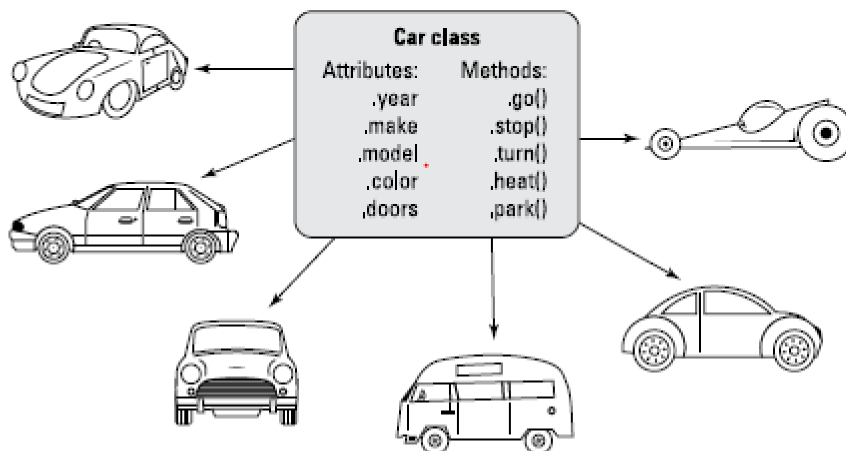
Robienie Pythona z klasą

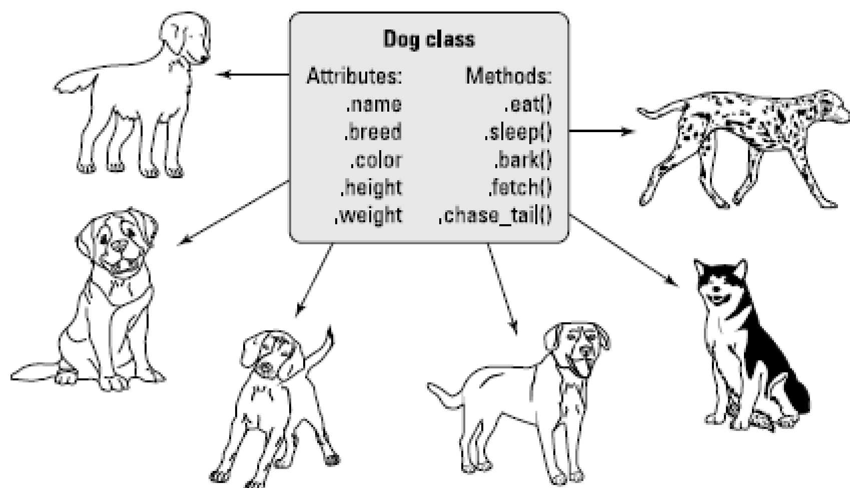
W poprzedniej części mówiliśmy o funkcjach, które pozwalają podzielić fragmenty kodu wykonujące określone zadania. W tej części dowiesz się o klasach, które pozwalają na podział kodu i danych. Odkrywasz cały cud, majestat i piękno klas i przedmiotów (no dobra, może trochę przeceniliśmy te rzeczy). Ale klasy stały się rozwijającą się cechą nowoczesnych obiektowych języków programowania, takich jak Python. Zdajemy sobie sprawę, że w poprzednich rozdziałach rzuciliśmy Ci mnóstwo techno żargonu. Nie martw się. Do końca tego rozdziału wychodzimy z założenia, że – podobnie jak 99,9 procent ludzi na tym świecie – nie odróżniasz klasy od przedmiotu z kanapki z pastrami.

Opanowanie klas i obiektów

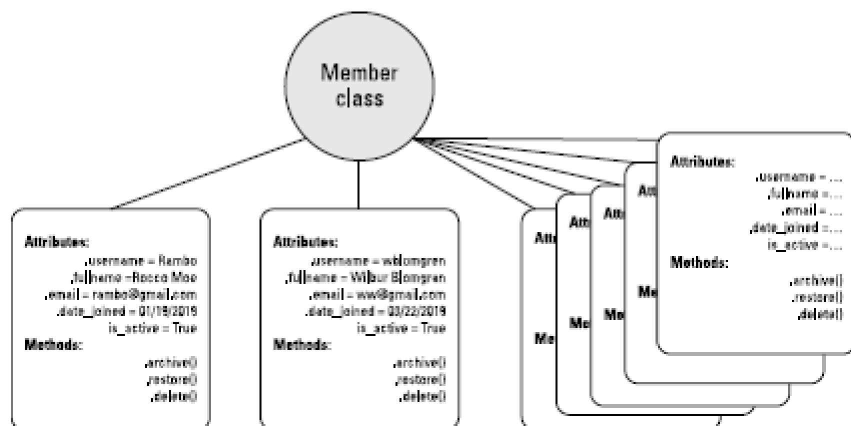
Programowanie zorientowane obiektowo (OOP) jest głównym modnym słowem w świecie komputerów od co najmniej kilku dekad. Termin obiekt wynika z faktu, że model przypomina przedmioty w prawdziwym słowie, ponieważ każdy przedmiot jest rzeczą, która ma pewne atrybuty i cechy, które czynią go wyjątkowym. Na przykład krzesło jest przedmiotem. Na świecie istnieje wiele różnych krzeseł, które różnią się rozmiarem, kształtem, kolorem i materiałem. Ale wszystkie nadal są krzesłami. A co z samochodami? Wszyscy rozpoznajemy samochód, gdy go widzimy. (Cóż, zazwyczaj.) Chociaż samochody nie są dokładnie takie same, wszystkie mają pewne cechy (rok, marka, model, kolor), które sprawiają, że każdy jest wyjątkowy. Mają pewne wspólne metody, gdzie metoda jest działaniem lub rzeczą, którą samochód może zrobić. Na przykład wszystkie samochody mają takie same akcje jazdy, zatrzymania i skrętu, którymi można sterować w każdym z nich. Rysunek



przedstawia koncepcję, w której wszystkie samochody (choć nie identyczne) mają pewne wspólne cechy i metody. W tym przypadku możesz myśleć o klasie „samochody” jako o fabryce, która tworzy wszystkie samochody. Po utworzeniu każdy samochód jest niezależnym obiektem. Zmiana jednego samochodu nie ma wpływu na inne samochody ani na klasę samochodu. Jeśli pomysł na fabrykę ci nie odpowiada, pomyśl o klasie jako o rodzaju planu. Weźmy na przykład psy. Nie, nie ma fizycznego planu tworzenia psów. Ale jest trochę psiego DNA, które prawie robi to samo. DNA psa można uznać za rodzaj planu (jak klasa Pythona), z którego tworzone są wszystkie psy. Psy różnią się cechami, takimi jak rasa, kolor i rozmiar, ale mają wspólne pewne zachowania (metody), takie jak „jedzenie” i „spanie”. Rysunek



pokazuje przykład, w którym istnieje klasa zwierząt o nazwie Pies, z której pochodzą wszystkie psy. W ten sposób nawet ludzi można łatwo postrzegać jako przedmioty. W swoim kodzie możesz utworzyć klasę Member, za pomocą której zarządzasz członkami swojej witryny. Każdy członek miałby określone atrybuty, takie jak nazwa użytkownika, imię i nazwisko itp. Możesz także mieć metody, takie jak .archive() do dezaktywacji konta i .restore() do ponownej aktywacji konta. Metody .archive() i .restore() to zachowania, które pozwalają kontrolować członkostwo w podobny sposób, w jaki akcelerator, hamulec i kierownica pozwalają kontrolować samochód. Rysunek przedstawia tę koncepcję



Najważniejsze jest to, że każda instancja klasy jest niezależnym obiektem, z którym można pracować. Zmiana jednej instancji klasy nie ma wpływu na klasę ani na inne instancje, podobnie jak pomalowanie jednego samochodu na inny kolor nie ma wpływu na fabrykę samochodów ani na żadne inne samochody produkowane przez tę fabrykę. Cały ten biznes klas i instancji wywodzi się z rodzaju programowania zwanego programowaniem obiektowym (w skrócie OOP). Jest to główna koncepcja w biznesie od kilku dekad. Python, jak każdy znaczący, poważny, nowoczesny język programowania, jest zorientowany obiektowo. Główne modne słowa, z którymi musisz się oswoić, to te, o których mówiłem w kilku ostatnich akapitach:

Klasa: Fragment kodu, z którego można wygenerować unikalny obiekt, gdzie każdy obiekt jest pojedynczą instancją klasy. Pomyśl o tym jak o planie lub fabryce, z której możesz tworzyć pojedyncze obiekty.

Instancja: Jedna jednostka danych plus kod wygenerowany z klasy jako instancja tej klasy. Każda instancja klasy jest również nazywana obiektem, tak jak wszystkie różne samochody są obiektami, wszystkie stworzone przez jakąś fabrykę samochodów (klasę).

Atrybut: Cecha obiektu, która zawiera informacje o obiekcie. Nazywana również właściwością obiektu. Nazwa atrybutu jest poprzedzona kropką, jak w przypadku member.username, które może zawierać nazwę użytkownika dla jednego członka witryny.

Metoda: funkcja języka Python powiązana z klasą. Definiuje akcję, którą obiekt może wykonać. W obiekcie metodę wywołuje się, poprzedzając nazwę metody kropką i poprzedzając ją parą nawiasów. Na przykład member.archive() może być metodą archiwizującą (dezaktywującą) konto członka.

Tworzenie klasy

Tworzysz własne klasy, tak jak tworzysz własne funkcje. Możesz nazwać klasę, jak chcesz, o ile jest to uzasadniona nazwa, która zaczyna się od litery i nie zawiera spacji ani znaków interpunkcyjnych. Zwyczajowo rozpoczyna się nazwę klasy wielką literą, aby ułatwić odróżnienie klas od zmiennych. Aby rozpocząć, wystarczy słowo klasa, po którym następuje spacja, wybrana nazwa klasy i dwukropek. Na przykład, aby utworzyć nową klasę o nazwie Członek, użyj klasy Członek: Aby kod był bardziej opisowy, możesz umieścić komentarz nad definicją klasy. Możesz także umieścić dokumentację pod linią klasy, która pojawi się za każdym razem, gdy wpiszesz nazwę klasy w VS Code. Na przykład, aby dodać komentarze do nowej klasy Member, możesz wpisać kod w następujący sposób:

Define a new class name Member.

class Member:

""" Create a new member. """

To tyle, jeśli chodzi o zdefiniowanie nowej klasy. Jednak nie jest to przydatne, dopóki nie określisz, jakie atrybuty mają dziedziczyć po klasie każdy obiekt tworzony z tej klasy.

PUSTE KLASY

Jeśli uruchomisz klasę z nazwą klasy:, a następnie uruchomisz swój kod przed zakończeniem klasy, w rzeczywistości pojawi się błąd. Aby to obejść, możesz powiedzieć Pythonowi, że po prostu nie jesteś jeszcze gotowy do napisania klasy, umieszczając słowo kluczowe pass pod definicją, tak jak w poniższym kodzie:

Define a new class name Member.

class Member:

pass

Zasadniczo to, co tam robisz, polega na mówieniu Pythonowi „Hej, wiem, że ta klasa jeszcze tak naprawdę nie działa, ale po prostu pozwól jej przejść i nie wyrzucaj komunikatu o błędzie z informacją o tym”.

Jak klasa tworzy instancję

Aby dać swojej klasie możliwość tworzenia instancji (obiektów) dla ciebie, dajesz klasie metodę init. Słowo init jest skrótem od inicjalizacji. Jako metoda jest to po prostu funkcja zdefiniowana wewnątrz klasy. Ale musi mieć specyficzną nazwę __init__ — to dwa podkreślenia, po których następuje init i

dwa kolejne podkreślenia. To __init__ jest czasami wymawiane jako „dunder init". Część dunder jest skrótem od podwójnego podkreślenia. Składnia tworzenia metody init jest następująca:

def __init__(self[, suppliedprop1, suppliedprop2,...])

Def to skrót od defne, a __init__ to nazwa wbudowanej metody Pythona, która może tworzyć obiekty z poziomu klasy. Część własna jest tylko nazwą zmiennej i jest używana w odniesieniu do tworzonego w danym momencie obiektu. Możesz użyć nazwy własnego wyboru zamiast siebie. Ale jaźń byłaby brana pod uwagę przez większość dobra „najlepsza praktyka", ponieważ jest wyjaśniająca i zwyczajowa. Cały ten biznes klas jest łatwiejszy do nauczenia się i zrozumienia, jeśli zaczniesz po prostu. Tak więc, dla przykładu roboczego, stwórzmy klasę o nazwie Member, do której będziesz przekazywać nazwę użytkownika (uname) i imię i nazwisko (fname) za każdym razem, gdy chcesz utworzyć członka. Jak zawsze kod można poprzedzić komentarzem. Możesz także umieścić dokument (w potrójnym cudzysłowie) pod pierwszym wierszem zarówno jako komentarz, jak i jako przypomnienie IntelliSense podczas wpisywania kodu w VS Code:

Define a class named Member for making member objects.

class Member:

""" Create a member from uname and fname """

def __init__(self, uname, fname):

Kiedy ta linia def __init__ jest wykonywana, masz pusty obiekt o nazwie self wewnątrz klasy. Parametry uname i fname po prostu przechowują wszelkie dane, które przekazujesz, a za chwilę zobaczysz, jak to działa. Pusty obiekt bez danych na niewiele się zda. To, co sprawia, że obiekt jest użyteczny, to zawarte w nim informacje, które są unikalne dla tego obiektu (jego atrybuty). Tak więc w twojej klasie następnym krokiem jest przypisanie wartości do każdego z atrybutów obiektu.

Nadawanie obiektowi jego atrybutów

Teraz, gdy masz nowy, pusty obiekt Member, możesz zacząć nadawać mu atrybuty i wypełniać (przechowywać wartości) te atrybuty. Załóżmy na przykład, że chcesz, aby każdy członek miał atrybut .username zawierający nazwę użytkownika (na przykład do logowania). Masz drugi atrybut o nazwie fullname, który jest pełnym imieniem i nazwiskiem członka. Aby zdefiniować i wypełnić te atrybuty, użyj

self.username = uname

self.fullname = fname

Pierwsza linia tworzy atrybut o nazwie username dla nowej instancji (self) i umieszcza w niej to, co zostało przekazane do atrybutu uname podczas wywoływania klasy. Druga linia tworzy atrybut o nazwie fullname dla nowego obiektu self i umieszcza w nim to, co zostało przekazane jako zmienna fname. Dodaj komentarz, a cała klasa może wyglądać tak:

Define a new class name Member.

class Member:

""" Create a new member. """

def __init__(self, uname, fname):

Define attributes and give them values.

self.username = uname

self.fullname = fname

Więc widzisz, co się tam dzieje? Linia __init__ tworzy nowy pusty obiekt o nazwie self. Następnie linia self.username = uname dodaje atrybut o nazwie username do tego pustego obiektu i umieszcza w tym atrybucie wszystko, co zostało przekazane jako uname. Następnie linia self.fullname = fname robi to samo z atrybutem fullname i przekazaną wartością fname. Konwencja nazewnictwa rzeczy w klasach sugeruje użycie początkowego ograniczenia dla nazwy klasy, ale atrybuty powinny być zgodne ze standardem dla zmiennych, pisanych małymi literami z podkreśleniem oddzielającym słowa w nazwie.

Tworzenie instancji z klasy

Po utworzeniu klasy możesz tworzyć z niej instancje (obiekty), używając tej prostej składni:

this_instance_name = Member('uname', 'fname')

Zamień tę_nazwę_instancji na wybraną przez siebie nazwę (w podobny sposób możesz nazwać psa, który jest instancją klasy dog). Zamień uname i fname na nazwę użytkownika i imię i nazwisko, które chcesz umieścić w obiekcie, który zostanie utworzony. Upewnij się, że nie wcinasz tego kodu; w przeciwnym razie Python pomyśli, że nowy kod nadal należy do kodu klasy. Tak nie jest. To nowy kod do testowania klasy. Tak więc, dla przykładu, powiedzmy, że chcesz utworzyć członka o nazwie new_guy z nazwą użytkownika Rambo i pełną nazwą Rocco Moe. Oto kod do tego:

new_guy = Member('Rambo','Rocco Moe')

Jeśli uruchomisz ten kod i nie otrzymasz żadnych komunikatów o błędach, to wiesz, że przynajmniej został uruchomiony. Ale żeby się upewnić, możesz wydrukować obiekt lub jego atrybuty. Aby więc zobaczyć, co naprawdę znajduje się w instancji new_guy Members, możesz wydrukować ją jako całość. Możesz także wydrukować tylko jego atrybuty, new_guy.username i new_guy.fullname. Możesz także wypisać type(new_guy), aby zapytać Pythona, czym jest „typ” new_guy. Oto ten kod:

print(new_guy)

print(new_guy.username)

print(new_guy.fullname)

print(type(new_guy))

Rysunek przedstawia cały kod i wynik działania go w komórce Jupytera.

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, uname, fname):
        # Define attributes and give them values.
        self.username = uname
        self.fullname = fname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

#See what's in the instance, as well as its individual properties.
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

```
<__main__.Member object at 0x000002175EA2E160>
Rambo
Rocco Moe
<class '__main__.Member'>
```

Na rysunku widać, że pierwszy wiersz danych wyjściowych wygląda następująco:

<__main__.Member object at 0x000002175EA2E160>

Mówi ci to, że new_guy jest obiektem utworzonym z klasy Member. Liczba na końcu to miejsce w pamięci, ale nie martw się o to; nie musisz o nich teraz wiedzieć. Następne trzy linie wyjścia to

Rambo

Rocco Moe

<class '__main__.Member'>

Linia Rambo to nazwa użytkownika new_guy (new_guy.username), Rocco Moe to pełne imię i nazwisko new_guy (new_guy.fullname). Typ to <class '__main__.Member'>, co znowu mówi ci tylko, że new_guy jest instancją klasy Member. Chociaż nie chcemy obciążać teraz komórek mózgowych, słowa przedmiot i własność są synonimami instancji i atrybutu. Instancję new_guy klasy Member można również nazwać obiektem, a atrybuty fullname i username klasy new_guy są również właściwościami tego obiektu. Trzeba przyznać, że na początku może być trochę trudno to ogarnąć, ale jest to naprawdę całkiem proste: obiekt to po prostu wygodny sposób na podzielenie informacji o przedmiocie, który jest podobny do innych przedmiotów (jak wszystkie psy to psy, a wszystkie samochody to samochody). To, co sprawia, że przedmiot jest wyjątkowy, to jego atrybuty, które niekoniecznie muszą być takie same jak atrybuty innych obiektów tego samego typu, podobnie jak nie wszystkie psy są tej samej rasy i nie wszystkie samochody są tego samego koloru . Celowo użyliśmy uname i fname jako nazw parametrów, aby odróżnić je od nazw atrybutów username i fullname. Nie jest to jednak wymagane. W rzeczywistości ludzie mają tendencję do używania tych samych nazw dla parametrów, co dla atrybutów. Zamiast uname jako symbol zastępczy, możesz użyć nazwy użytkownika (nawet jeśli jest taka sama jak nazwa atrybutu). Podobnie, możesz użyć pełnej nazwy zamiast fname. Nie zmieni to zachowania klasy. Musisz tylko pamiętać, że ta sama nazwa jest używana na dwa różne sposoby, najpierw jako symbol zastępczy danych przekazywanych do klasy, a później jako rzeczywista nazwa atrybutu, która uzyskuje swoją wartość od przekazanej wartości. Rysunek pokazuje ten sam kod, co na rysunku 6-4, z uname zastąpionym nazwą użytkownika i fname zastąpioną pełną nazwą. Uruchomienie

kodu daje dokładnie takie same dane wyjściowe jak poprzednio; używanie tej samej nazwy dla dwóch różnych rzeczy ani trochę nie przeszkadzało Pythonowi.

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

#See what's in the instance, as well as its individual properties.
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

```
<__main__.Member object at 0x000002175EA2E240>
Rambo
Rocco Moe
<class '__main__.Member'>
```

Po wpisaniu nazwy klasy i otwierającego nawiasu w VS Code, jego IntelliSense pokazuje składnię parametrów i pierwszy ciąg dokumentów w kodzie, jak pokazano na rysunku. Nazywanie rzeczy w zrozumiały dla ciebie sposób i umieszczanie w klasie opisowej dokumentacji ułatwia zapamiętanie, jak korzystać z klasy w przyszłości.

```
7           self.full   Member(self, username, fullname)
8
9     # The class ends    param username
10
11    # Create an insta   Create a new member.
12    new_guy = Member(
```

Zmiana wartości atrybutu

Pracując z krotkami, możesz definiować pary klucz:wartość, podobnie jak pary atrybut:wartość, które widzisz tutaj z instancjami klasy. Istnieje jednak jedna zasadnicza różnica: krotki są niezmienne, co oznacza, że po ich zdefiniowaniu kod nie może nic w nich zmienić. Nie dotyczy to przedmiotów. Po utworzeniu obiektu możesz w dowolnym momencie zmienić wartość dowolnego atrybutu za pomocą metody i następującej prosta składnia:

objectname.attributename = value

Zamień nazwę obiektu na nazwę obiektu (który już utworzyłeś za pomocą klasy). Zamień nazwę atrybutu na nazwę atrybutu, którego wartość chcesz zmienić. Zastąp wartość nową wartością. Na rysunku pokazano przykład, w którym po początkowym utworzeniu obiektu new_guy wykonywany jest następujący wiersz kodu:

new_guy.username = "Princess"

Linie danych wyjściowych poniżej pokazują, że nazwa użytkownika new_guy rzeczywiście została zmieniona na Princess. Jego pełne imię i nazwisko nie uległo zmianie, ponieważ nie zrobiłeś nic z tym w swoim kodzie.

Definiowanie atrybutów z wartościami domyślnymi

Nie musisz przekazywać wartości każdego atrybutu dla nowego obiektu. Jeśli zawsze zamierzasz nadać im jakąś domyślną wartość w momencie tworzenia obiektu, możesz po prostu użyć self.attributename = value, tak samo jak poprzednio, w którym nazwaatrybutu jest wybraną przez ciebie nazwą. Wartością może być wartość, którą właśnie ustawiłeś, na przykład Prawda lub Fałsz dla wartości logicznej, dzisiejsza data lub cokolwiek innego, co Python może obliczyć lub określić bez podawania mu wartości. Załóżmy na przykład, że za każdym razem, gdy tworzysz nowego członka, chcesz śledzić datę utworzenia tego członka w atrybucie o nazwie date_joined. I chcesz mieć możliwość aktywowania i dezaktywowania kont w celu kontrolowania logowania użytkowników. Więc tworzysz atrybut o nazwie is_active. Załóżmy, że chcesz utworzyć nowego członka z ustawieniem True. Jeśli zamierzasz robić cokolwiek z datami i godzinami, będziesz chciał zaimportować moduł datetime, więc umieść go na początku pliku, nawet przed wierszem class Member:. Następnie możesz dodać te linie przed lub po innych liniach, które przypisują wartości atrybutom w klasie:

```
self.date_joined = dt.date.today()

self.is_active = True
```

Oto jak możesz dodać import i te dwa nowe atrybuty do klasy:

```
import datetime as dt

# Define a new class name Member.

class Member:

""" Create a new member. """

def __init__(self, username, fullname):

# Define attributes and give them values.

self.username = username

self.fullname = fullname

# Default date_joined to today's date.

self.date_joined = dt.date.today()

# Set is active to True initially.

self.is_active = True
```

Jeśli zapomnisz zaimportować datetime na początku kodu, po uruchomieniu kodu pojawi się komunikat o błędzie informujący, że nie wie, co oznacza dt.date.today(). Po prostu dodaj wiersz importu na początku kodu i spróbuj ponownie. Nie ma potrzeby przekazywania żadnych nowych danych do klasy dla atrybutów date_joined i is_active, ponieważ można je określić za pomocą kodu. Zauważ, że wartość domyślna to po prostu: Jest to wartość, która jest przypisywana automatycznie podczas pierwszego tworzenia obiektu. Ale to nie znaczy, że nie można tego zmienić. Możesz zmienić te wartości tak samo, jak zmieniasz wartość dowolnego innego atrybutu za pomocą składni

objectname.attributename = value

Załóżmy na przykład, że używasz atrybutu is_active do określenia, czy użytkownik jest aktywny i czy może zalogować się do Twojej witryny. Jeśli członek okaże się wstrętnym trollem i nie chcesz, aby się już logował, możesz po prostu zmienić atrybut is_active na False w następujący sposób:

newmember.is_active = False

TRWAŁE ZMIANY DANYCH

Być może zastanawiasz się, jaki jest sens tworzenia tych wszystkich różnych klas i obiektów, jeśli wszystko po prostu przestaje istnieć w momencie zakończenia programu. Co to znaczy utworzyć „członka", jeśli nie można przechowywać tych informacji „na zawsze" i używać ich do kontrolowania logowania członków do witryny internetowej lub czegokolwiek innego? Prawdę mówiąc, nic ci to nie da. . . samodzielnie. Jednak wszystkie dane, które tworzysz i zarządzasz z klasami i obiektami mogą być utrwalane (przechowywane w nieskończoność) i być do Twojej dyspozycji w dowolnym momencie, przechowując te dane w jakimś zewnętrznym pliku, zwykle w bazie danych. Do kwestii utrwalania danych dochodzimy w księdze 3 tej książki. Ale najpierw naprawdę musisz nauczyć się podstawowych podstaw Pythona, ponieważ zrozumienie, jak to wszystko działa, jest prawie niemożliwe, jeśli nie rozumiesz, jak to wszystko działa.

Podanie metod klasowych

Każdy zdefiniowany obiekt może mieć dowolną liczbę atrybutów, z których każdy może mieć dowolną nazwę, w celu przechowywania informacji o obiekcie, takich jak rasa i kolor psa lub marka i model samochodu. Możesz także zdefiniować własne metody dla dowolnego obiektu, które bardziej przypominają zachowania niż fakty dotyczące obiektu. Na przykład pies może jeść, spać i szczekać. Samochód może jechać, zatrzymywać się i skręcać. Metoda jest tak naprawdę tylko funkcją, o której dowiedziałeś się w poprzednim rozdziale. To, co czyni ją metodą, to fakt, że jest powiązana z określoną klasą iz każdym konkretnym obiektem, który tworzysz w tej klasie. Nazwy metod różnią się od nazw atrybutów obiektu parą nawiasów, które występują po nazwie. Aby zdefiniować, jakie metody będą w twojej klasie, użyj następującej składni dla każdej metody:

def methodname(self[, param1, param2, ...])

Zamień nazwę metody na wybraną nazwę (wszystkie małe litery, bez spacji). Zachowaj tam słowo self jako odniesienie do obiektu definiowanego przez klasę. Możesz także przekazać parametry po self, używając przecinków, tak jak w przypadku każdej innej funkcji, ale jest to całkowicie opcjonalne. Nigdy nie wpisuj nawiasów kwadratowych ([]). Są one pokazane tutaj w składni tylko po to, aby wskazać, że nazwy parametrów po self są dozwolone, ale nie wymagane. Stwórzmy metodę o nazwie .show_date_joined(), która zwraca nazwę użytkownika i datę połączenia w sformatowanym łańcuchu. Oto jak możesz napisać ten kod, aby zdefiniować tę metodę:

Define methods as functions, use self for "this" object.

def show_datejoined(self):

return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"

Nazwa metody to show_datejoined. Zadaniem tej metody, gdy zostanie wywołana, jest po prostu złożenie ładnie sformatowanego tekstu zawierającego pełne imię i nazwisko członka oraz datę dołączenia. Upewnij się, że wcięcie def jest na tym samym poziomie co pierwsza def, ponieważ nie można ich wciąć pod atrybutami.

Aby wywołać metodę z kodu, użyj tej składni:

objectname.methodname()

Zastąp nazwę obiektu nazwą obiektu, do którego się odwołujesz. Zamień nazwę metody na nazwę metody, którą chcesz wywołać. Dołącz nawiasy (bez spacji) i pozostaw je puste, jeśli wewnątrz klasy jedynym parametrem między nawiasami jest parametr self. Rysunek przedstawia pełny przykład. Zwróć uwagę, jak na rysunku 6.8 zdefiniowano metodę show_datejoined() w klasie.

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # Methods for each instance created (instance methods)

    # A method to return a formatted string with showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# Try out the date_joined method.
print(new_guy.show_datejoined())

Rocco Moe joined on 12/06/18
```

Jego def jest wcięty do tego samego poziomu pierwszego def. Kod, który wykonuje metoda, jest wcięty pod tym. Poza klasą new_guy = Member('romo', 'Rocco Moe') tworzy nowego członka o nazwie new_guy. Następnie new_guy.show_datejoined() wykonuje metodę show_datejoined(), która z kolei wyświetla Rocco Moe dołączył 12/06/18, w dniu, w którym uruchomiłem kod.

Przekazywanie parametrów do metod

Możesz przekazywać dane do metod w ten sam sposób, w jaki robisz funkcje, używając nazw parametrów w nawiasach. Należy jednak pamiętać, że self zawsze pojawia się tam jako pierwsze i nigdy nie otrzymuje danych z zewnątrz. Załóżmy na przykład, że chcesz utworzyć metodę o nazwie .activate() i ustawić jej wartość True, jeśli użytkownik może się zalogować, lub False, gdy użytkownik nie ma takiej możliwości. Wszystko, co przekażesz, jest przypisane do atrybutu .is_active. Oto jak zdefiniować tę metodę w swoim kodzie:

Method to activate (True) or deactivate (False) account.

def activate(self, yesno):

""" True for active, False to make inactive """

self.is_active = yesno

Docstring, który tam umieściliśmy, jest opcjonalny. Ale pojawia się na ekranie, gdy wpisujesz odpowiedni kod w VS Code, więc służy jako dobre przypomnienie o tym, co możesz przekazać. Po uruchomieniu ta metoda nie pokazuje niczego na ekranie, po prostu zmienia is_active atrybut tego elementu członkowskiego do tego, co przekazałeś jako parametr yesno. Pomaga zrozumieć, że metoda jest tak naprawdę tylko funkcją. To, co odróżnia metodę od funkcji generycznej, to fakt, że metoda jest zawsze powiązana z jakąś klasą. Więc to nie jest tak ogólne jak funkcja. Rysunek 6.9 przedstawia całą klasę, a następnie kod do jej testowania. Linia new_guy = Member('romo', 'Rocco Moe') tworzy nowy obiekt członkowski o nazwie new_guy. Następnie print(new_guy.is_active) wyświetla wartość atrybutu is_active, która ma wartość True, ponieważ jest to wartość domyślna dla wszystkich nowych członków.

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # Methods for each instance created (instance methods)

    # A method to return a formatted string with showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"

    # Method to activate (True) or deactive (False) account.
    def activate(self, yesno):
        """ True for active, False to make inactive """
        self.is_active = yesno

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# Is new-guy active?
print(new_guy.is_active)

# Try out the activate method.
new_guy.activate(False)

# Is new-guy still active?
print(new_guy.is_active)

True
False
```

Linia new_guy.activate(False) wywołuje metodę aktywacji() dla tego obiektu i przekazuje jej wartość logiczną False. Następnie print(new_guy.is_active) dowodzi, że wywołanie aktywacji rzeczywiście zmieniło atrybut is_active dla new_guy z True na False.

Wywołanie metody klasy według nazwy klasy

Jak już widziałeś, możesz wywołać metodę klasy za pomocą składni

specificobject.method()

Alternatywą jest użycie określonej nazwy klasy, co może pomóc uczynić kod nieco łatwiejszym do zrozumienia dla człowieka. Nie ma dobrego ani złego sposobu, najlepszej ani najgorszej praktyki, aby to zrobić. Istnieją tylko dwa różne sposoby osiągnięcia celu i możesz użyć tego, który wolisz. W każdym razie ta alternatywna składnia to:

Classname.method(specificobject)

Zamień Classname na nazwę klasy (którą zwykle definiujemy zaczynając od dużej litery), po której następuje nazwa metody, a następnie umieść konkretny obiekt (który prawdopodobnie już utworzyłeś) w nawiasach. Załóżmy na przykład, że tworzymy nowego członka o imieniu Wilbur, używając klasy Member i tego kodu:

wilbur = Member('wblomgren', 'Wilbur Blomgren')

Tutaj wilbur jest konkretnym obiektem, który utworzyliśmy z klasy Member. Możemy wywołać metodę show_datejoined() na tym obiekcie, korzystając ze składni, którą już znasz, na przykład:

print(wilbur.show_datejoined())

Alternatywą jest wywołanie metody show_datejoined() klasy Member i przekazanie jej określonego obiektu, wilbur, w następujący sposób:

print(Member.show_datejoined(wilbur))

Dane wyjściowe z obu metod są dokładnie takie same, jak w poniższym (ale z datą uruchomienia kodu):

Wilbur Blomgren dołączył 12.06.18

Ponownie, ta druga metoda nie jest szybsza, wolniejsza, lepsza, gorsza ani nic podobnego. Jest to po prostu alternatywna składnia, której możesz użyć, a niektórzy ludzie ją preferują, ponieważ rozpoczęcie wiersza od Member wyjaśnia, do której klasy należy metoda show_datejoined() . To z kolei może sprawić, że kod będzie bardziej czytelny dla innych programistów lub dla ciebie za rok, jeśli nie będziesz pamiętał żadnej z rzeczy, które pierwotnie napisałeś w tej aplikacji.

Korzystanie ze zmiennych klasy

Do tej pory widziałeś przykłady atrybutów, które są czasami nazywane zmiennymi instancji, ponieważ są symbolami zastępczymi zawierającymi informacje, które różnią się w zależności od instancji klasy. Na przykład w klasie psów rasa psa może być dla jednego psa Pudel, a dla innego Sznaucer. Istnieje inny typ zmiennej, którego można używać z klasami, zwany zmienną klasową, która jest stosowana do wszystkich nowych wystąpień klasy, które nie zostały jeszcze utworzone. Zmienne klasy wewnątrz klasy nie mają żadnego powiązania z self, ponieważ słowo kluczowe self zawsze odnosi się do konkretnego tworzonego w danym momencie obiektu. Aby zdefiniować zmienną klasy, umieść wskaźnik myszy nad linią def __init__ i zdefiniuj zmienną, używając standardowej składni:

variablename = value

Zamień nazwę zmiennej na wybraną przez siebie nazwę i zastąp wartość określoną wartością, którą chcesz przypisać tej zmiennej. Załóżmy na przykład, że w zgłoszeniu Twojego członka znajduje się kod, który zapewnia ludziom trzymiesięczny (90 dni) bezpłatny dostęp po rejestracji. Nie jesteś pewien, czy chcesz związać się z tym na zawsze, więc zamiast zakodować go na stałe w swojej aplikacji (więc trudno go zmienić), możesz po prostu ustawić go jako zmienną klasową, która będzie automatycznie stosowana do wszystkich nowych obiektów, na przykład :

Define a class named Member for making member objects.

class Member:

""" Create a member object """

free_days = 90

def __init__(self, username, fullname):

Ta zmienna free_days jest definiowana przed zdefiniowaniem __init__, więc nie jest powiązana z żadnym konkretnym obiektem w kodzie. Następnie, powiedzmy, później w kodzie, który chcesz zapisać datę wygaśnięcia bezpłatnego okresu próbnego. Możesz mieć atrybuty o nazwach date_joined i free_expires, które reprezentują dzisiejszą datę plus liczbę dni określoną przez free_days. Intuicyjnie może się wydawać, że możesz dodać wolne_dni do daty, używając prostej składni, takiej jak ta:

self.free_expires = dt.date.today() dt.timedelta(days = free_days)

To by nie zadziałało. Jeśli spróbujesz uruchomić kod w ten sposób, pojawi się błąd mówiący, że Python nie rozpoznaje nazwy zmiennej free_days (mimo że jest ona zdefiniowana na samym początku klasy). Aby to zadziałało, musisz poprzedzić nazwę zmiennej nazwą klasy lub self. Na przykład to zadziała:

self.free_expires = dt.date.today() dt.timedelta(days = Member.free_days)

Rysunek przedstawia większy obraz.

```
import datetime as dt
# Define a new class name Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date
        self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
wilbur = Member('wblomgren', 'Wilbur Blomegren')

print(wilbur.date_joined)
print(wilbur.free_expires)

2018-12-06
2019-12-06
```

Usunęliśmy część kodu z oryginalnej klasy, aby go skrócić i ułatwić skupienie się na nowych rzeczach. Linia free_days = 365 u góry ustawia wartość zmiennej free_days na 365. Następnie w dalszej części kodu zastosowano metodę Member.free_freedays, aby dodać tę liczbę dni do bieżącej daty. Uruchomienie tego kodu poprzez utworzenie nowego członka o imieniu wilbur i wyświetlenie jego atrybutów date_joined i free_expires pokazuje aktualną datę (niezależnie od tego, gdzie siedzisz, kiedy uruchamiasz kod) oraz datę 365 dni później. A co, jeśli później zdecydujesz, że dawanie ludziom 90 darmowych dni to dużo. Możesz po prostu zmienić to bezpośrednio w klasie, ale ponieważ jest to zmienna, możesz to zrobić w locie, na przykład poza klasą:

#Set a default for free days.

Member.free_days = 90

Po uruchomieniu tego kodu nadal tworzysz użytkownika o nazwie wilbur ze zmiennymi date_joined i free_days. Ale tym razem wilbur.free_expires upłynie 90 dni po połączeniu, a nie 365 dni

Korzystanie z metod klasowych

Przypomnij sobie, że metoda jest funkcją powiązaną z określoną klasą. Do tej pory metody, których używałeś, takie jak .show_datejoined() i .activate() były metodami instancji, ponieważ zawsze używasz ich z określonym obiektem. . . konkretna instancja klasy. W Pythonie możesz także tworzyć metody klasowe. Jak sama nazwa wskazuje, metoda klasowa to metoda powiązana z klasą jako całością, a nie z konkretnymi instancjami klasy. Innymi słowy, metody klasowe mają podobny zakres do zmiennych klasowych, ponieważ dotyczą całej klasy, a nie tylko poszczególnych instancji klasy. Podobnie jak w przypadku zmiennych klasowych, nie potrzebujesz słowa kluczowego self z metodami klasowymi, ponieważ to słowo kluczowe zawsze odnosi się do konkretnego obiektu tworzonego w danym momencie, a nie do wszystkich obiektów tworzonych przez klasę. Więc na początek, jeśli chcesz, aby metoda zrobiła coś z klasą jako całością, nie używaj def name(self), ponieważ self natychmiast wiąże metodę z jednym obiektem. Byłoby miło, gdyby do stworzenia metod klasowych wystarczyło tylko wykluczyć słowo self, ale niestety tak to nie działa. Aby zdefiniować metodę klasy, musisz najpierw wpisać to do swojego kodu:

classmethod

Znak @ na początku definiuje go jako dekoratora — tak, to kolejne modne słowo, które można dodać do stale rosnącej listy modnych frazesów typu nerd-o-rama. Dekorator to na ogół coś, co zmienia lub rozszerza funkcjonalność tego, do czego jest stosowany. Pod tym wierszem zdefiniuj metodę klasy, używając następującej składni:

def methodname(cls,x):

Zamień nazwę metody na dowolną nazwę, którą chcesz nadać metodzie. Pozostaw cls bez zmian, ponieważ jest to odniesienie do klasy jako całości (ponieważ dekorator @classmethod zaprojektował to jako takie za kulisami). Po cls możesz umieścić przecinki i nazwy parametrów, które chcesz przekazać metodzie, tak samo jak w przypadku zwykłych metod instancji. Załóżmy na przykład, że chcesz zdefiniować metodę ustawiającą liczbę dni wolnych tuż przed rozpoczęciem tworzenia obiektów, tak aby wszystkie obiekty miały tę samą liczbę dni_wolnych. Poniższy kod realizuje to, najpierw definiując zmienną klasową o nazwie free_days, która ma zadaną wartość domyślną równą zero (wartość domyślna może być dowolna). W dalszej części klasy znajduje się ta metoda klasowa:

Class methods follow @classmethods and use cls rather then self.

@classmethod

def setfreedays(cls,days):

cls.free_days = days

Ten kod mówi Pythonowi, że gdy ktoś wywołuje metodę setfreedays() w tej klasie, powinien ustawić wartość cls.free_days (zmienna klasy free_days dla tej klasy) na dowolną liczbę dni, które upłynęły.

Rysunek przedstawia kompletny przykład w komórce Jupytera (którą z pewnością możesz wpisać i wypróbować samodzielnie) oraz wyniki uruchomienia tego kodu.

```python
import datetime as dt
# Define a new class name Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date
        self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)

    # Class methods follow @classmethod decorator and refer to cls rather then to self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days
```

Zobaczmy więc, co się stanie, gdy uruchomisz ten kod. Ta linia:

Member.setfreedays(30)

... mówi Pythonowi, aby wywołał metodę setfreedays() klasy Python i przekazał jej liczbę 30. Tak więc wewnątrz klasy zmienna free_days = 0 otrzymuje nową wartość 30, zastępując oryginalne 0. Łatwo o tym zapomnieć wielkie i małe litery mają duże znaczenie w Pythonie, zwłaszcza że wydaje się, że używasz małych liter przez 99,9 procent czasu. Ale z reguły nazwy klas zaczynają się od początkowej litery, więc każde wywołanie nazwy klasy musi również zaczynać się od początkowej litery. Następnie kod tworzy członka o imieniu wilbur, a następnie wypisuje wartości jego atrybutów date_joined i free_expires:

wilbur = Member('wblomgren', 'Wilbur Blomgren')

print(wilbur.date_joined)

print(wilbur.free_expires)

Dokładny wynik zależy od daty uruchomienia tego kodu. Jednak pierwsza data powinna być datą dzisiejszą, podczas gdy data free_expires powinna przypadać 30 dni później (lub dowolną liczbę dni określoną w wierszu Member.setfreedays(30)).

Używanie metod statycznych

Właśnie wtedy, gdy myślałeś, że w końcu skończyłeś uczyć się o klasach, okazuje się, że istnieje inny rodzaj metody, którą możesz stworzyć w klasie Pythona. Nazywa się to metodą statyczną i zaczyna się od tego dekoratora: @staticmethod. Więc przynajmniej ta część jest łatwa. Tym, co odróżnia metodę statyczną od metod instancji i klas, jest to, że metoda statyczna nie odnosi się konkretnie do instancji obiektu ani nawet do klasy jako całości. To naprawdę jest funkcja ogólna i tak naprawdę jedynym powodem zdefiniowania jej jako części klasy byłaby chęć użycia tej samej nazwy w innym miejscu w kodzie. Innymi słowy, ściśle organizacyjnym zadaniem jest trzymanie razem kodu, który pasuje do siebie, aby łatwo go było znaleźć, gdy przeglądasz kod w celu zmiany lub upiększenia rzeczy. W każdym razie pod tą linią @staticmethod definiujesz swoją metodę statyczną tak samo jak każdą inną metodę, ale nie używasz self ani cls. Ponieważ metoda statyczna nie jest ściśle powiązana z klasą lub obiektem, z wyjątkiem zakresu, w jakim chcesz ją tam zachować w celu uporządkowania kodu. Oto przykład:

@staticmethod

def currenttime():

now = dt.datetime.now()

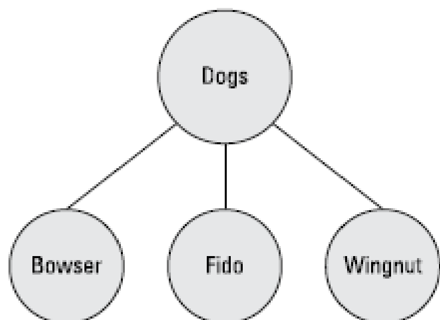
return f"{now:%I:%M %p}"

Mamy więc metodę o nazwie currenttime(), która nie oczekuje przekazywania żadnych danych i nie dba nawet o obiekt, z którym pracujesz, ani nawet o klasę; po prostu pobiera bieżącą datę i godzinę za pomocą now = dt.datetime.now() , a następnie zwraca te informacje w ładnym formacie typu 12:00 PM. Rysunek przedstawia pełny przykład, w którym można zobaczyć, jak metoda statyczna jest odpowiednio wcięta i wpisana pod koniec klasy. Gdy kod spoza klasy wywołuje metodę Member.currenttime(), posłusznie zwraca niezależnie od tego, jaka jest godzina , nawet bez mówienia czegokolwiek o konkretnym obiekcie z tej klasy.

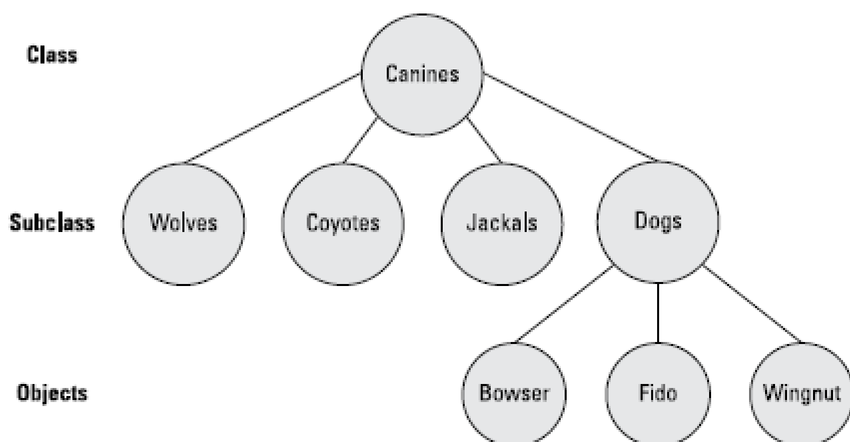
```
import datetime as dt

# Define a class named Member for making member objects.
class Member:
    # This is a class variable that's the same for all instances.
    free_days = 0

    """ Create a member object from username and fullname """
    def __init__(self, username, fullname):
        # Define properties and assign default values.
        self.datejoined = dt.date.today()
        self.free_expires = dt.date.today() + dt.timedelta(Member.free_days)

    # Class methods follow @classmethods and use cls rather then self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days

    @staticmethod
    def currenttime():
        now = dt.datetime.now()
        return f"{now:%I:%M %p}"

# Class definition ends at last indented line

# Try out the new static method (no object required)
print(Member.currenttime())

03:24 PM
```

Zrozumienie dziedziczenia klas

Ludzie, którzy naprawdę interesują się programowaniem obiektowym, żyją po to, by rozmawiać o dziedziczeniu klas i podklasach itd., a to niewiele lub nic nie znaczy dla przeciętnego Joe czy Josephine z ulicy. Mimo to to, o czym mówią jako o koncepcji Pythona, jest w rzeczywistości czymś, co cały czas widzisz w prawdziwym życiu. Jak wspomniano wcześniej, jeśli uznamy, że DNA psa jest rodzajem „fabryki" lub klasy Pythona, możemy połączyć wszystkie psy jako członków klasy zwierząt, którą nazywamy psami. Chociaż każdy pies jest wyjątkowy, wszystkie psy nadal są psami, ponieważ należą do klasy, którą nazywamy psami, i możemy to zilustrować na rysunku .



Tak więc każdy pies jest wyjątkowy (chociaż żaden inny pies nie jest tak dobry jak twój), ale to, co sprawia, że psy są do siebie podobne, to cechy, które dziedziczą po klasie psów. Pojęcia klas i dziedziczenia klas, które oferuje Python i inne języki zorientowane obiektowo, nie pojawiły się z jasnego nieba tylko po to, by uczynić naukę trudniejszą i bardziej irytującą. Wiele informacji na świecie można najlepiej przechowywać, kategoryzować i rozumieć za pomocą klas, podklas i podklas, aż do jednostek. Na przykład, być może zauważyłeś, że po planecie wędrują inne podobne do psów stworzenia (chociaż prawdopodobnie nie są to te, które chciałbyś trzymać w domu jako zwierzęta domowe). Przychodzą na myśl wilki, kojoty i szakale. Są podobne do psów w tym, że wszystkie dziedziczą swoją psiość po klasie wyższego poziomu, którą moglibyśmy nazwać psami, jak pokazano na rysunku



Korzystając z naszej analogii z psem, z pewnością nie musimy zatrzymywać się przy kłach w drodze na górę. Możemy umieścić ssaki wyżej, ponieważ wszystkie psy są ssakami. Możemy umieścić zwierzęta ponad tym, ponieważ wszystkie ssaki są zwierzętami. I możemy postawić żywe istoty ponad to, ponieważ wszystkie zwierzęta są żywymi istotami. Tak więc zasadniczo wszystko, co czyni psa psem, wynika z faktu, że każdy z nich dziedziczy pewne cechy z wielu „klas” lub stworzeń, które go poprzedzały. Dla mózgowców biologii, tak, wiem, że Mammalia to klasa, Canis to rodzaj, a poniżej są gatunki. Nie musisz więc wysyłać mi e-maili ani wiadomości w tej sprawie. Używam tutaj terminów klas i podklas tylko po to, aby powiązać tę koncepcję z klasami, podklasami i obiektami w Pythonie. Oczywiście ta koncepcja nie dotyczy tylko psów. Na świecie jest też wiele różnych kotów. Jest urocza mała Bootsy, z którą chętnie dzielisz łóżko, i mnóstwo innych kotów, takich jak lwy, tygrysy i jaguary, z którymi prawdopodobnie nie chciałbyś. Jeśli wygooglujesz hierarchię żywych istot i klikniesz Obrazy, zobaczysz, ile jest sposobów klasyfikowania wszystkich żywych istot i jak działa dziedziczenie od ogólnej do konkretnej żywej istoty. Nawet nasza analogia do samochodu może podążać za tym. Na górze mamy

pojazdy transportowe. Pod tym być może łodzie, samoloty i samochody. Pod samochodami mamy samochody, ciężarówki, furgonetki i tak dalej, i tak dalej, aż do jednego konkretnego samochodu. Tak więc klasy i podklasy nie są niczym nowym. Nowością jest po prostu myślenie o przedstawieniu tych rzeczy bezmyślnym maszynom, które nazywamy komputerami. Zobaczmy więc, jak byś to zrobił. Z perspektywy kodowania najłatwiejszym sposobem dziedziczenia jest tworzenie podklas w obrębie klasy. Klasa definiuje rzeczy, które mają zastosowanie do wszystkich instancji tej klasy. Każda podklasa definiuje rzeczy, które są istotne tylko dla podklasy, nie zastępując niczego, co pochodzi z ogólnej klasy „rodzic”.

Tworzenie klasy bazowej (głównej).

Podklasy dziedziczą wszystkie atrybuty i metody jakiejś klasy głównej wyższego poziomu lub klasy nadrzędnej, która jest zwykle nazywana klasą podstawową. Ta klasa jest po prostu dowolną klasą, niczym nie różniącą się od tego, co widziałeś do tej pory w tym rozdziale. Ponownie użyjemy klasy Members, ale ograniczymy ją do kilku podstawowych elementów, które nie mają nic wspólnego z podklasami, więc nie będziesz musiał przekopywać się przez cały dodatkowy, nieistotny kod. Oto podstawowa klasa:

```
import datetime as dt

# Class is used for all kinds of people.

import datetime as dt

# Base class is used for all kinds of Members.

class Member:

""" The Member class attributes and methods are for everyone """

# By default, a new account expires in one year (365 days)

expiry_days = 365

# Initialize a member object.

def __init__(self, firstname, lastname):

# Attributes (instance variables) for everybody.

self.firstname = firstname

self.lastname = lastname

# Calculate expiry date from today's date.

self.expiry_date = dt.date.today() dt.timedelta(days=self.expiry_days)
```

Domyślnie nowe konta wygasają po roku. Tak więc ta klasa najpierw ustawia zmienną klasy o nazwie „dni_wygaśnięcia” na 365, która będzie używana w późniejszym kodzie do obliczenia daty wygaśnięcia na podstawie dzisiejszej daty. Jak zobaczysz później, użyliśmy zmiennej klasy, aby to zdefiniować, ponieważ możemy nadać jej nową wartość z podklasy. Aby przykładowy kod był prosty i przejrzysty, ta wersja klasy Member akceptuje tylko dwa parametry, imię i nazwisko. Rysunek przedstawia przykład testowania kodu z hipotetycznym członkiem o imieniu Joe.

```
import datetime as dt
# Class is used for all kinds of people.
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)

# Outside the class now.
Joe = Member('Joe', 'Anybody')
print(Joe.firstname)
print(Joe.lastname)
print(Joe.expiry_date)

    Joe
    Anybody
    2019-12-08
```

Wydrukowanie imienia, nazwiska i daty wygaśnięcia Joe pokazuje, czego można się spodziewać po klasie, która przekazuje imię Joe i nazwisko Anybody. Kiedy uruchamiasz kod, data_wygaśnięcia powinna wynosić jeden rok od dowolnej daty, jaką nam podałeś, kiedy uruchomiłeś kod. Załóżmy teraz, że naszym prawdziwym zamiarem jest stworzenie dwóch różnych rodzajów użytkowników, administratorów i użytkowników. Oba typy użytkowników będą miały atrybuty oferowane przez klasę Member. Tak więc definiując te typy użytkowników jako podklasy Member, automatycznie uzyskają oni te same atrybuty (i metody, jeśli takie istnieją).

Definiowanie podklasy

Aby zdefiniować podklasę, upewnij się, że kursor znajduje się pod klasą bazową iz powrotem bez wcięcia, ponieważ podklasa nie jest częścią ani nie jest zawarta w klasie podstawowej. Aby zdefiniować podklasę, użyj następującej składni:

class subclassname(mainclassname):

Zastąp nazwę podklasy dowolną nazwą tej podklasy. Zamień mainclassname na nazwę klasy bazowej, zgodnie z definicją na górze klasy bazowej. Na przykład, aby utworzyć podklasę Member o nazwie Admin, użyj:

class Admin(Person):

Aby utworzyć kolejną podklasę o nazwie Użytkownik, dodaj ten kod:

class User(Person):

Jeśli pozostawisz puste klasy, nie będziesz mógł testować, ponieważ pojawi się komunikat o błędzie informujący, że klasa jest pusta. Ale możesz umieścić słowo pass jako pierwsze polecenie w każdym z nich. W ten sposób możesz powiedzieć Pythonowi „Tak, wiem, że te klasy są puste, ale pozwól temu przejść, nie wyrzucaj komunikatu o błędzie”). Możesz umieścić komentarz

nad każdym z nich, aby przypomnieć, do czego służy każdy z nich, tak jak poniżej:

Subclass for Admins.

class Admin(Member):

pass

Subclass for Users.

class User(Member):

pass

Używając podklas, nie musisz bezpośrednio odwoływać się do klasy Member. Administratorzy i Użytkownicy automatycznie odziedziczą wszystkie wpisy członkowskie. Na przykład, aby utworzyć administratora o imieniu Annie, użyjesz następującej składni:

Ann = Admin('Annie', 'Angst')

Aby utworzyć użytkownika, zrób to samo z klasą użytkownika i nazwą użytkownika. Na przykład:

Uli = User('Uli', 'Ungula')

Aby sprawdzić, czy ten kod działa, możesz zrobić to samo, co dla Member Joe. Po utworzeniu dwóch kont użyj instrukcji print(), aby zobaczyć, co się na nich znajduje. Rysunek przedstawia wyniki tworzenia dwóch użytkowników.

```
# Subclass for Admins.
class Admin(Member):
    pass

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst')
print(Ann.firstname)
print(Ann.lastname)
print(Ann.expiry_date)
print()
Uli = User('Uli', 'Ungula')
print(Uli.firstname)
print(Uli.lastname)
print(Uli.expiry_date)

    Annie
    Angst
    2019-12-03

    Uli
    Ungula
    2019-12-03
```

Ann jest Administratorem, a Uli Użytkownikiem, ale oboje automatycznie otrzymują wszystkie atrybuty (atrybuty) przypisane członkom. (Klasa Member znajduje się bezpośrednio nad kodem pokazanym na obrazku. Pominąłem to, ponieważ się nie zmieniło). Nauczyłeś się tutaj, że podklasa akceptuje wszystkie różne parametry akceptowane przez klasę podstawową i przypisuje je do atrybutów, tak samo jak klasa Person. Ale jak dotąd Administrator i Użytkownik są tylko członkami bez unikalnych cech. W prawdziwym życiu prawdopodobnie będą pewne różnice między tymi dwoma typami użytkowników. W następnych sekcjach nauczysz się różnych sposobów, aby te różnice się urzeczywistniły.

Zastępowanie wartości domyślnej z podklasy

Jedną z najprostszych rzeczy, które możesz zrobić z podklasą, jest nadanie atrybutowi, który ma wartość domyślną w klasie bazowej, inną wartość. Na przykład w klasie Member utworzyliśmy zmienną o nazwie data_wygaśnięcia, która będzie używana później w klasie do obliczenia daty ważności. Ale załóżmy, że chcesz, aby konta administratora nigdy nie wygasały (lub wygasały po jakimś absurdalnym czasie, więc wciąż jest tam jakaś data). Wszystko, co musisz zrobić, to ustawić nową datę wygaśnięcia w klasie Admin (i możesz usunąć linię pass, ponieważ klasa nie będzie już pusta). Oto jak może to wyglądać w Twojej podklasie Admin:

Subclass for Admins.

class Admin(Member):

Admin accounts don't expire for 100 years.

expiry_days = 365.2422 * 100

Jakakolwiek wartość, którą przekażesz, zastąpi domyślny zestaw u góry klasy Member i zostanie użyta do obliczenia daty wygaśnięcia administratora.

Dodawanie dodatkowych parametrów z podklasy

Czasami członkowie podklasy mają pewną wartość parametru, której nie mają inni członkowie. W takim przypadku możesz chcieć przekazać parametr z podklasy, który nawet nie istnieje w klasie podstawowej. Robienie tego jest trochę bardziej skomplikowane niż zmiana wartości domyślnej, ale jest to dość powszechna technika, więc powinieneś być tego świadomy. Przeanalizujmy przykład. Na początek twoja podklasa będzie potrzebować własnej linii def __init__, która zawiera wszystko, co znajduje się w __init__ klasy bazowej, plus wszelkie dodatkowe rzeczy, które chcesz przechodzić. Załóżmy na przykład, że administratorzy mają jakiś tajny kod, który chcesz przekazać z podklasy Admin. Nadal musisz podać imię i nazwisko, więc twoja linia def __init__ w podklasie Admin będzie wyglądać tak:

def __init__(self, firstname, lastname, secret_code):

Poziom wcięcia będzie taki sam jak linie nad nim. Następnie wszystkie parametry, które należą do klasy bazowej Member, muszą zostać tam przekazane przy użyciu tej dość dziwnie wyglądającej składni:

super().__init__(param1, param2,...)

Zastąp param1, param2 itd. nazwami parametrów, które chcesz wysłać do klasy bazowej. Powinno to być wszystko, co jest już w parametrach Członka, z wyłączeniem siebie. W tym przykładzie Członek oczekuje tylko imienia i nazwiska. Tak więc kod dla tego przykładu będzie wyglądał następująco:

super().__init__(firstname, lastname)

To, co zostanie, możesz przypisać do obiektu podklasy, używając standardowej składni:

self.parametername = parametername

Zamień parametr nazwa_parametru na nazwę parametru, którego nie wysłałeś do Członka. W tym przypadku byłby to parametr secret_code. Więc kod byłby:

self.secret_code = secret_code

Rysunek przedstawia przykład, w którym utworzyliśmy użytkownika Admin o imieniu Ann i przekazaliśmy PRESTO jako jej tajny kod. Wydrukowanie wszystkich jej atrybutów pokazuje, że nadal

ma właściwą datę ważności i tajny kod. Jak widać, stworzyliśmy również zwykłego Użytkownika o imieniu Uli. Dane Uli nie mają żadnego wpływu na zmiany administratora.

```
# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100
    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)

print()
Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date)

Annie Angst 2118-12-03 PRESTO

Uli Ungula 2019-12-03
```

Pozostała jedna mała luka, która ma związek z faktem, że Użytkownik nie ma tajnego kodu. Więc jeśli spróbujesz wydrukować .secret_code dla osoby, która ma konto użytkownika, a nie konto administratora, otrzymasz komunikat o błędzie. Jednym ze sposobów radzenia sobie z tym jest po prostu pamiętanie, że Użytkownicy nie mają tajnych kodów i nigdy nie próbują uzyskać do nich dostępu. Alternatywnie możesz przekazać użytkownikom tajny kod, który jest tylko pustym ciągiem znaków. Więc kiedy próbujesz go wydrukować lub wyświetlić, nic nie dostajesz, ale nie pojawia się też komunikat o błędzie. Aby użyć tej metody, po prostu dodaj to do głównej klasy Member:

```
# Domyślny tajny kod to nic
self.secret_code = ""
```

Więc nawet jeśli nie robisz nic z secret_code w podklasie User, nie musisz się martwić o błąd podczas próby uzyskania dostępu do tajnego kodu dla użytkownika. Użytkownik będzie miał tajny kod, ale będzie to tylko pusty ciąg znaków. Rysunek przedstawia cały kod z obiema podklasami, a także próbę wydrukowania kodu Uli.secret_code, który po prostu niczego nie wyświetla bez zgłaszania komunikatu o błędzie.

```
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class properties and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Properties (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
        # Default secret code is nothing
        self.secret_code = ''

    # Method in the base class
    def showexpiry(self):
        return  f"{self.firstname} {self.lastname} expires on {self.expiry_date}"

# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100

    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)
print() # Add a blank line to output.

Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date, Uli.secret_code)
```

```
Annie Angst 2118-12-08 PRESTO

Uli Ungula 2019-12-08
```

Opuściliśmy podklasę User z pass jako jedyną instrukcją. W prawdziwym życiu prawdopodobnie wymyśliłbyś więcej wartości domyślnych lub parametrów dla swoich innych podklas. Ale składnia i kod są dokładnie takie same dla wszystkich podklas, więc nie będziemy się nad tym rozwodzić. Umiejętności, których nauczyłeś się w tej sekcji, będą działać dla wszystkich twoich klas i podklas.

Wywołanie metody klasy bazowej

Metody w klasie bazowej działają tak samo dla podklas, jak i dla klasy podstawowej. Aby to wypróbować, dodaj nową metodę o nazwie showexpire(self) na końcu klasy bazowej w następujący sposób:

class Member:

""" The Member class attributes and methods are for everyone """

By default, a new account expires in one year (365 days)

expiry_days = 365

Initialize a member object.

def __init__(self, firstname, lastname):

Attributes (instance variables) for everybody.

self.firstname = firstname

self.lastname = lastname

Calculate expiry date from today's date.

self.expiry_date = dt.date.today() dt.timedelta(days=self.expiry_days)

Default secret code is nothing

self.secret_code = ''

Method in the base class

def showexpiry(self):

return f"{self.firstname} {self.lastname} expires on {self.

expiry_date}"

Metoda showexpiry() po wywołaniu zwraca sformatowany ciąg zawierający imię i nazwisko użytkownika oraz datę wygaśnięcia. Pozostawienie nietkniętych podklas i wykonanie kodu wyświetla imiona i daty wygaśnięcia Ann i Uli:

Ann = Admin('Annie', 'Angst', 'PRESTO')

print(Ann.showexpiry())

Uli = User('Uli', 'Ungula')

print(Uli.showexpiry())

Oto dane wyjściowe, chociaż daty będą się różnić w zależności od daty uruchomienia kodu:

Annie Angst expires on 2118-12-04

Uli Ungula expires on 2019-12-04

Dwukrotne użycie tej samej nazwy

Jedynym luźnym końcem, nad którym możesz się zastanawiać, jest to, co się stanie, gdy użyjesz tej samej nazwy więcej niż raz? Python zawsze wybierze najbardziej specyficzną, powiązaną z podklasą. Użyje bardziej ogólnej metody z klasy podstawowej tylko wtedy, gdy w podklasie nie ma nic, co ma tę nazwę metody. Aby to zilustrować, oto kod, który redukuje klasę Member do zaledwie kilku atrybutów i metod, aby usunąć z drogi nieistotny kod. Komentarze w kodzie opisują, co się dzieje w kodzie:

class Member:

""" The Member class attributes and methods """

Initialize a member object.

def __init__(self, firstname, lastname):

Attributes (instance variables) for everybody.

self.firstname = firstname

self.lastname = lastname

```
# Method in the base class

def get_status(self):

return f"{self.firstname} is a Member."

# Subclass for Administrators

class Admin(Member):

def get_status(self):

return f"{self.firstname} is an Admin."

# Subclass for regular Users

class User(Member):

def get_status(self):

return f"{self.firstname} is a regular User."
```

Klasa Member oraz klasa Admin i User mają metodę o nazwie get_status(), która pokazuje imię i status członka. Rysunek przedstawia wynik uruchomienia tego kodu z administratorem, użytkownikiem i członkiem, który nie jest ani administratorem, ani użytkownikiem.

```
class Member:
    """ The Member class attributes and methods are for everyone """
    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname

    # Method in the main class
    def get_status(self):
        return f"{self.firstname} is a Member."

# Subclass for Administrators
class Admin(Member):
    def get_status(self):
        return f"{self.firstname} is an Admin."

# Subclass for regular Users
class User(Member):
    def get_status(self):
        return f"{self.firstname} is a regular User."

# Create an admin
Ann = Admin('Annie', 'Angst')
print(Ann.get_status())

#Create a user
Uli = User('Uli', 'Ungula')
print(Uli.get_status())

# create a member (neither Admin or user)
Manny = Member("Mindy", "Membo")
print(Manny.get_status())

Annie is an Admin.
Uli is a regular User.
Mindy is a Member.
```

Jak widać, get_status wywoływana w każdym przypadku to get_status(), która jest powiązana z podklasą użytkownika (lub klasą bazową w przypadku osoby, która nie jest Adminem ani Użytkownikiem). Python ma wbudowaną metodę help(), której można użyć z dowolną klasą, aby

uzyskać więcej informacji o tej klasie. Na przykład na dole kodu z rysunku 6.19 dodaj następujący wiersz:

help(Admin)

Gdy ponownie uruchomisz kod, zobaczysz informacje o tej klasie Admin, jak widać na rysunku.

```
help(Admin)

Help on class Admin in module __main__:

class Admin(Member)
 |  Admin(firstname, lastname)
 |
 |  The Member class attributes and methods are for everyone
 |
 |  Method resolution order:
 |      Admin
 |      Member
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  get_status(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Member:
 |
 |  __init__(self, firstname, lastname)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Member:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Nie musisz się teraz martwić wszystkimi szczegółami tej figury, więc nie martw się, jeśli jest trochę onieśmielająca. Na razie najważniejsza jest sekcja zatytułowana Method Resolution Order, która wygląda tak:

Method resolution order:

Admin

Member

builtins.object

Kolejność rozpoznawania metod mówi ci, że jeśli klasa (i jej podklasy) mają metody o tej samej nazwie (np. metody i użyć tej metody, jeśli istnieje. Jeśli metoda get_status() nie została zdefiniowana w podklasie Admin, to szuka ona w klasie Member i używa tej metody, jeśli zostanie znaleziona. Jeśli żadna z nich nie miała metody get_status, to szuka w buildins.object, która jest odniesieniem do pewnych wbudowanych metod, które są wspólne dla wszystkich klas i podklas. Tak więc najważniejsze jest to, że jeśli przechowujesz swoje dane w hierarchiach klas i podklas i wywołujesz metodę na podklasie, użyje ona tej metody podklasy, jeśli istnieje. Jeśli nie, użyje metody klasy bazowej, jeśli istnieje. Jeśli to również nie istnieje, wypróbuje wbudowane metody. A jeśli wszystko inne zawiedzie, zgłosi błąd ponieważ nie może znaleźć metody, którą twój kod próbuje wywołać. Zwykle główną przyczyną tego typu błędów jest po prostu błędna nazwa metody w kodzie, przez co Python nie może

jej znaleźć. Przykładem wbudowanej metody jest __dict__. Dyktat jest skrótem od słownika, a są to podwójne podkreślenia otaczające skrót. Wracając do rysunku 6-20, wykonanie polecenia

print(Admin.__dict__)

. . . nie powoduje błędu, mimo że nigdy nie zdefiniowaliśmy metody o nazwie __dict__. Dzieje się tak dlatego, że istnieje wbudowana metoda o tej nazwie, która po wywołaniu z print() pokazuje słownik metod (zarówno twoich, jak i wbudowanych) dla tego obiektu. To naprawdę nie jest coś, w co musisz się zbytnio angażować na tak wczesnym etapie nauki. Pamiętaj tylko, że jeśli spróbujesz wywołać metodę, która nie istnieje na żadnym z tych trzech poziomów, na przykład:

print(Admin.snookums())

... pojawia się błąd, który wygląda mniej więcej tak:

---> print(Admin.snookums())

AttributeError: type object 'Admin' has no attribute 'snookums'

To mówi ci, że Python nie ma pojęcia, o co chodzi w snookums(), więc może tylko zgłosić błąd. W prawdziwym życiu tego rodzaju błąd jest zwykle spowodowany po prostu błędną pisownią nazwy metody w kodzie. Klasy (i do pewnego stopnia podklasy) są dość często używane w świecie Pythona, a to, czego się tu nauczyłeś, powinno stosunkowo ułatwić pisanie własnych klas, a także zrozumienie klas napisanych przez innych. Zanim skończymy tę książkę, warto zapoznać się z bardziej „podstawową" koncepcją Pythona, a mianowicie sposób, w jaki Python obsługuje błędy i rzeczy, które możesz zrobić we własnym kodzie, aby lepiej radzić sobie z błędami.