**Spieranie się o większe fragmenty kodu**

Tu dowiesz się, jak lepiej zarządzać większymi projektami kodu, tworząc własne funkcje. Funkcje umożliwiają podział kodu na małe zadania, które można wywoływać z wielu miejsc w aplikacji. Na przykład, jeśli coś, do czego potrzebujesz dostępu w całej aplikacji, wymaga tuzina wierszy kodu, prawdopodobnie nie chcesz powtarzać tego kodu w kółko za każdym razem, gdy tego potrzebujesz. Spowoduje to, że kod będzie większy niż powinien. Ponadto, jeśli chcesz coś zmienić lub jeśli musisz naprawić błąd w tym kodzie, nie chcesz robić tego wielokrotnie w wielu różnych miejscach. Gdyby cały ten kod był zawarty w funkcji, wystarczyłoby zmienić lub naprawić go w tej jednej lokalizacji. Aby uzyskać dostęp do zadania, które wykonuje funkcja, po prostu wywołujesz tę funkcję ze swojego kodu. Robisz to dokładnie w taki sam sposób, jak wywołujesz funkcję wbudowaną, taką jak print: po prostu wpisujesz nazwę w swoim kodzie. Możesz także wymyślić własne nazwy funkcji. Pomyśl więc o funkcjach jako o sposobie personalizacji języka Python, tak aby jego polecenia pasowały do tego, czego potrzebujesz w swojej aplikacji.

**Tworzenie funkcji**

Tworzenie funkcji jest łatwe. W rzeczywistości możesz śledzić tutaj w komórce notatnika Jupyter lub pliku .py, jeśli chcesz zdobyć praktyczne doświadczenie. Praktyka naprawdę pomaga w zrozumieniu i zapamiętywaniu rzeczy. Aby utworzyć funkcję, rozpocznij nową linię od def (skrót od definicji), po której następuje spacja, a następnie wybrana nazwa, po której następuje para nawiasów bez spacji przed lub w środku. Następnie umieść dwukropek na końcu tej linii. Na przykład, aby utworzyć prostą funkcję o nazwie hello(), wpisz:

```
def hello():
```

To jest funkcja. Jednak to nic nie daje. Aby funkcja coś zrobiła, musisz napisać kod w Pythonie, który podpowiada jej, co ma robić w kolejnych wierszach. Aby upewnić się, że nowy kod znajduje się „wewnątrz” funkcji, wprowadź wcięcie w każdym z tych wierszy. Wcięcia liczą się bardzo długo w Pythonie. Nie ma polecenia oznaczającego koniec funkcji. Wszystkie wcięte linie poniżej linii def są częścią tej funkcji. Pierwsza linia bez wcięcia (z wcięciem sięgającym linii def) znajduje się poza funkcją. Aby ta funkcja coś zrobiła, musisz umieścić wcięty wiersz kodu pod def. Zaczniemy łatwo, po prostu mając funkcję przywitaj się. Więc wpisz print('Hello') z wcięciem pod linią def. Teraz twój kod wygląda tak:

```
def hello():

print('Hello')
```

Jeśli teraz uruchomisz kod, nic się nie stanie. To jest w porządku. Nic nie powinno się wydarzyć, ponieważ kod wewnątrz funkcji nie jest wykonywany, dopóki funkcja nie zostanie wywołana. Własne funkcje wywołujesz w ten sam sposób, w jaki wywołujesz funkcje wbudowane, pisząc kod, który wywołuje funkcję według nazwy, łącznie z nawiasami na końcu. Na przykład, jeśli podążasz dalej, naciśnij Enter, aby dodać pustą linię, a następnie wpisz hello() (bez spacji) i upewnij się, że nie jest wcięty. (Nie chcesz, aby ten kod był wcięty, ponieważ wywołuje funkcję w celu wykonania jej kodu; nie jest częścią funkcji). Wygląda to tak:

```
def hello():

print('Hello')

hello()
```

Mimo to nic się nie dzieje, jeśli jesteś w komórce Jupytera lub pliku .py, ponieważ do tej pory wpisałeś tylko kod. Aby cokolwiek się stało, musisz uruchomić kod w zwykły sposób w Jupyter lub VS Code (jeśli używasz pliku .py w VS Code). Po wykonaniu kodu powinieneś zobaczyć wynik, który jest po prostu słowem Hello, jak pokazano na rysunku


**Komentowanie funkcji**

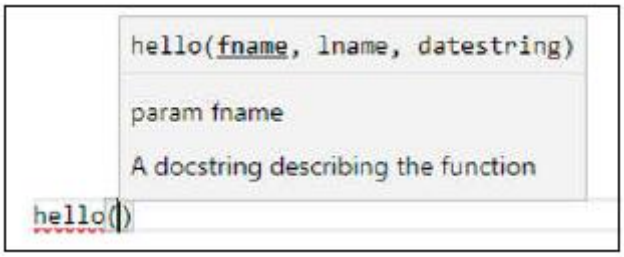
Komentarze są zawsze opcjonalne w kodzie. Ale jest to dość zwyczajowe, aby pierwszy wiersz pod instrukcją def był docstringiem (tekst ujęty w potrójne cudzysłowy), który opisuje, co robi funkcja. Dość często umieszcza się komentarz poprzedzony znakiem # po prawej stronie nawiasów w pierwszym wierszu. Oto przykład użycia prostej funkcji hello():

def hello(): # Practice function

""" A docstring describing the function """

print('Hello')

Ponieważ są to tylko komentarze, nie mają żadnego wpływu na działanie kodu. Ponowne uruchomienie kodu wyświetla dokładnie te same wyniki. Dzieje się tak dlatego, że komentarze są tylko notatkami dla siebie lub członków zespołu programistycznego opisującymi, o co chodzi w kodzie. Jako dodatkowy bonus dla użytkowników VS Code, gdy zaczniesz wpisywać nazwę funkcji, pomoc IntelliSense VS Code wyświetli instrukcję def dla Twojej niestandardowej funkcji, jak również wpisany dla niej dokument, jak na rysunku. Możesz więc stworzyć własną niestandardową pomoc dla własnych niestandardowych funkcji.




**Przekazywanie informacji do funkcji**

Możesz przekazywać informacje funkcjom, aby mogły nad nimi pracować. Aby to zrobić, wprowadź nazwę parametru dla każdej informacji, którą będziesz przekazywać do funkcji. Nazwa parametru jest taka sama jak nazwa zmiennej. Możesz sam to wymyślić. Używaj małych liter, bez spacji i znaków interpunkcyjnych. Idealnie, parametr powinien opisywać, co jest przekazywane, aby kod był czytelny, ale jeśli wolisz, możesz użyć nazw ogólnych, takich jak x i y. Każda nazwa podana jako parametr jest lokalna tylko dla tej funkcji. Na przykład, jeśli masz zmienną o nazwie x poza funkcją i inną zmienną o nazwie x, która znajduje się wewnątrz funkcji, zmiany wprowadzone w x wewnątrz funkcji nie wpłyną

na zmienną o nazwie x, która jest poza funkcją. Technicznym terminem określającym sposób, w jaki zmienne działają wewnątrz funkcji, jest zasięg lokalny, co oznacza, że zakres istnienia i wpływu zmiennych pozostaje wewnątrz funkcji i nie rozciąga się dalej. Zmienne, które są tworzone i modyfikowane wewnątrz funkcji, dosłownie przestają istnieć w momencie, gdy funkcja przestaje działać, a wszelkie zmienne, które zostały zdefiniowane poza funkcją, nie mają wpływu na to, co dzieje się wewnątrz funkcji. To dobrze, ponieważ pisząc funkcję, nie musisz się martwić, że przypadkowo zmienisz zmienną poza funkcją, która ma taką samą nazwę. Funkcja może zwrócić wartość, a zwrócona wartość jest widoczna poza funkcją. Więcej o tym, jak to wszystko działa, za chwilę. Załóżmy na przykład, że chcesz, aby funkcja hello pozdrawiała każdego, kto korzysta z aplikacji (i masz dostęp do tych informacji w jakiejś zmiennej). Aby przekazać informacje do funkcji i użyć ich tam:

* Umieść nazwę w nawiasach funkcji, aby działała jako symbol zastępczy dla przychodzących informacji.

* Wewnątrz funkcji użyj dokładnie tej samej nazwy, aby pracować z przekazywanymi informacjami.

Załóżmy na przykład, że chcesz przekazać imię osoby do funkcji hello, a następnie użyć go w instrukcji print(). Możesz użyć dowolnej nazwy ogólnej dla obu, na przykład:

```
def hello(x): # Practice function

""" A docstring describing the function """

print('Hello ' + x)
```

Nazwy ogólne nie ułatwiają zrozumienia kodu. Więc prawdopodobnie lepiej byłoby użyć bardziej opisowej nazwy, takiej jak nazwa lub nawet nazwa_użytkownika, jak poniżej:

```
def hello(user_name): # Practice function

""" A docstring describing the function """

print('Hello ' +  user_name)
```

W funkcji print() dodaliśmy spację po o w Hello, aby między Hello a nazwą w danych wyjściowych była spacja. Gdy funkcja ma parametr, musisz przekazać jej wartość, gdy ją wywołujesz, inaczej nie będzie działać. Na przykład, jeśli dodałeś parametr do instrukcji def i nadal próbowałeś wywołać funkcję bez parametru, jak w poniższym kodzie, uruchomienie kodu spowodowałoby błąd:

```
def hello(user_name): # Practice function

""" A docstring describing the function """

print('Hello ' + user_name)

hello()
```

Dokładny błąd odczytałby coś w stylu hello() brak 1 wymaganego argumentu pozycyjnego: „nazwa_użytkownika”, co jest głównym sposobem na powiedzenie, że funkcja hello oczekuje, że coś zostanie do niej przekazane. W przypadku tej konkretnej funkcji należy przekazać ciąg znaków. Wiemy o tym, ponieważ łączymy wszystko, co jest przekazywane do zmiennej, z innym łańcuchem (słowo hello, po którym następuje spacja). Jeśli spróbujesz połączyć liczbę z łańcuchem, pojawi się błąd. Wartość, którą przekazujesz, może być dosłowna (dokładna data, którą chcesz przekazać) lub może to być nazwa zmiennej zawierającej te informacje. Na przykład, gdy uruchomisz ten kod:

```
def hello(user_name): # Practice function
```

""" A docstring describing the function """

print('Hello ' + user_name)

hello('Alan')

. . . wynik to Hello Alan, ponieważ podałeś Alan jako ciąg znaków w hello('Alan'), więc imię Alan jest zawarte w danych wyjściowych. Możesz także użyć zmiennej do przekazywania danych. Na przykład w poniższym kodzie przechowujemy ciąg „Alan” w zmiennej o nazwie this_person. Następnie wywołujemy funkcję, używając tej nazwy zmiennej. Tak więc uruchomienie tego kodu generuje Hello Alan, jak pokazano na rysunku 5-3.

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

# Put a string in a variable named this_person.
this_person = 'Alan'
# Pass that variable name to the function.
hello(this_person)
```

```
Hello Alan
```

**Definiowanie parametrów opcjonalnych z wartościami domyślnymi**

W poprzedniej sekcji wspomnieliśmy, że gdy wywołujesz funkcję, która „oczekuje” parametrów bez przekazywania tych parametrów, pojawia się błąd. To było małe kłamstwo. Możesz napisać funkcję, w której przekazanie parametru jest opcjonalne, ale musisz powiedzieć jej, czego ma użyć, jeśli nic nie zostanie przekazane. Składnia tego jest następująca:

def functioname(paremetername = defaultvalue):

Jedyną rzeczą, która naprawdę się różni, jest część = defaultvalue po nazwie parametru. Na przykład możesz przepisać naszą przykładową funkcję hello() z wartością domyślną, taką jak ta:

def hello(user_name = 'nobody'): # Practice function

""" A docstring describing the function """

print('Hello ' + user_name)

Rysunek przedstawia funkcję po dokonaniu tej zmiany. Ten kod wywołał również funkcję, zarówno z parametrem, jak i bez. Jak widać na wyjściu, nie ma błędu. Pierwsze wywołanie wyświetla cześć, po której następuje przekazana nazwa. Puste wywołanie funkcji hello() powoduje wykonanie funkcji z domyślną nazwą użytkownika, nikt, więc wynikiem jest Witaj nikt.

```
def hello(user_name = 'nobody'):    # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello('Alan')
hello()
```

```
Hello Alan
Hello nobody
```

**Przekazywanie wielu wartości do funkcji**

Do tej pory we wszystkich naszych przykładach przekazywaliśmy do funkcji tylko jedną wartość. Ale możesz przekazać tyle wartości, ile chcesz. Po prostu podaj nazwę parametru dla każdej wartości i oddziel nazwy przecinkami. Załóżmy na przykład, że chcesz przekazać do funkcji imię, nazwisko i być może datę użytkownika. Możesz zdefiniować te trzy parametry w następujący sposób:

def hello(fname, lname, datestring): # Practice function

""" A docstring describing the function """

print('Hello ' + fname ' +  ' lname)

print('The date is ' + datestring)

Zauważ, że żaden z parametrów nie jest opcjonalny. Więc podczas wywoływania funkcji musisz przekazać trzy wartości, takie jak ta:

hello('Alan', 'Simpson', '12/31/2019')

Rysunek przedstawia ten kod i dane wyjściowe po przekazaniu trzech wartości.

```
def hello(fname, lname, datestring):    # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
```

```
Hello Alan Simpson you mentioned 12/31/2019
```

Jeśli chcesz użyć niektórych (ale nie wszystkich) parametrów opcjonalnych z wieloma parametrami, upewnij się, że te opcjonalne są wprowadzone jako ostatnie. Rozważmy na przykład następujące, które nie działałoby poprawnie:

def hello(fname, lname='unknown', datestring):

Jeśli spróbujesz uruchomić ten kod z takim układem, pojawi się błąd, który odczytuje coś w rodzaju SyntaxError: non-default argument następuje po argumencie domyślnym. Próbuje ci powiedzieć, że jeśli chcesz wyświetlić w funkcji zarówno parametry wymagane, jak i parametry opcjonalne, musisz najpierw wprowadzić wszystkie wymagane (w dowolnej kolejności). Następnie opcjonalne parametry mogą być wymienione po tym z ich znakami = (w dowolnej kolejności). Więc to by zadziałało:

def hello(fname, lname, datestring=''):

msg = 'Hello ' + fname ' + ' lname

if len(datestring) > 0:

msg = ' you mentioned ' +  datestring

print(msg)

**Logicznie kod wewnątrz funkcji mówi:**

Utwórz zmienną o nazwie message i wstaw Hello oraz imię  i nazwisko. Jeśli przekazany ciąg daty ma długość większą niż zero, dodaj „wspomniałeś” i ten ciąg daty do tej zmiennej msg.

Wydrukuj wszystko, co jest w zmiennej msg w tym momencie. Na rysunku 5.6 przedstawiono dwa przykłady wywołania tej wersji funkcji.

```
def hello(fname, lname, datestring=''):
    msg = 'Hello ' + fname + ' ' + lname
    if len(datestring) > 0:
        msg += ' you mentioned ' + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
hello('Sammy', 'Schmeedledorp')
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Sammy Schmeedledorp
```

Pierwsze wywołanie przekazuje trzy wartości, a drugie wywołanie przekazuje tylko dwie. Oba działają, ponieważ trzeci parametr jest opcjonalny. Dane wyjściowe z pierwszego wywołania to pełne dane wyjściowe, w tym data. Drugi pomija część dotyczącą daty w danych wyjściowych.

**Używanie argumentów słów kluczowych (kwargs)**

Jeśli kiedykolwiek przeglądałeś oficjalną dokumentację Pythona na Python.org, być może zauważyłeś, że często używa się terminu kwargs. To skrót od argumentów słów kluczowych i jest kolejnym sposobem przekazywania danych do funkcji. Termin argument jest terminem technicznym określającym „wartość, którą przekazujesz do parametrów funkcji”. Do tej pory używaliśmy argumentów stricte pozycyjnych. Rozważmy na przykład te trzy parametry:

def hello(fname, lname, datestring=''):

Kiedy wywołujesz funkcję w ten sposób:

hello("Alan", "Simpson")

Python „zakłada” „Alan” to imię, ponieważ jest to pierwszy przekazany argument, a fname to pierwszy parametr w funkcji. Zakłada się, że drugim argumentem „Simpson” jest lname, ponieważ lname jest drugim parametrem w instrukcji def. Zakłada się, że ciąg daty jest pusty, ponieważ ciąg daty jest trzecim parametrem w instrukcji def i nic nie jest przekazywane jako trzeci parametr. To nie musi być w ten sposób. Jeśli chcesz, możesz powiedzieć funkcji, co jest czym, używając parametru składni = wartość w kodzie wywołującym funkcję. Na przykład spójrz na to wezwanie do cześć:

hello(datestring='12/31/2019', lname='Simpson', fname='Alan')

Gdy uruchomisz ten kod, działa on dobrze, mimo że kolejność przekazywanych argumentów nie zgadza się z kolejnością nazw parametrów w instrukcji def. Ale kolejność nie ma tutaj znaczenia, ponieważ parametr, z którym związany jest każdy argument, jest dołączany do wywołania. Oczywiście argument „Alan” idzie w parze z parametrem fname, ponieważ fname to nazwa parametru w instrukcji def. Działa, jeśli przekazujesz również zmienne. Ponownie, kolejność nie ma znaczenia. Oto inny sposób, aby to zrobić:

attpt_date = '12/30/2019'

last_name = 'Janda'

first_name = 'Kylie'

hello(datestring=apt_date, lname=last_name, fname=first_name)

Rysunek  przedstawia wynik działania kodu w obie strony.

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

# Pass in in literal kwargs (identify each by parameter name)
hello(datestring='12/31/2019', lname='Simpson', fname='Alan')

# Pass in in kwargs from variables (identify each by parameter name)
appt_date = '12/30/2019'
last_name = 'Janda'
first_name = 'Kylie'
hello(datestring=appt_date, lname=last_name, fname=first_name)
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Kylie Janda you mentioned 12/30/2019
```

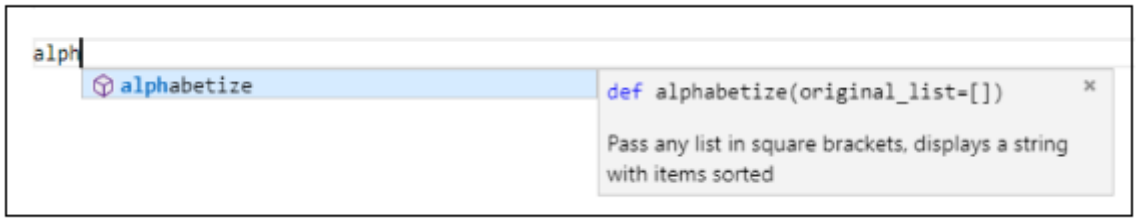
Jak widać, wszystko działa dobrze, ponieważ nie ma dwuznaczności co do tego, który argument pasuje do którego parametru, ponieważ nazwa parametru jest określona w kodzie wywołującym.

**Przekazywanie wielu wartości na liście**

Do tej pory przekazywaliśmy pojedynczo dane. Ale możesz także przekazać iteracje do funkcji. Pamiętaj, że iterowalność to wszystko, przez co Python może przechodzić w pętli, aby uzyskać wartości. Lista jest prostym i prawdopodobnie najczęściej używanym literałem. To tylko kilka wartości oddzielonych przecinkami ujętych w nawiasy kwadratowe. Główną sztuczką w pracy z listami jest to, że jeśli chcesz zmienić zawartość listy w jakikolwiek sposób (na przykład poprzez posortowanie zawartości listy), musisz utworzyć kopię listy w funkcji i wprowadzić zmiany w tej kopii . Dzieje się tak dlatego, że funkcja nie otrzymuje oryginalnej listy w zmiennym (zmiennym) formacie, tylko otrzymuje „wskaźnik” do listy, co jest zasadniczo sposobem na poinformowanie jej, gdzie znajduje się lista, aby mogła pobrać jej zawartość. Z pewnością może uzyskać te treści, ale nie może ich zmienić na tej oryginalnej liście. Nie jest to duży problem, ponieważ możesz posortować dowolną listę za pomocą prostej metody sort() (lub sort(reverse=True), jeśli chcesz posortować w kolejności malejącej). Na przykład oto nowa funkcja o nazwie alfabetyzacja(), która przyjmuje jeden argument o nazwie nazwy. Zakłada się, że wewnątrz tej funkcji znajduje się lista wartości. Tak więc funkcja może uporządkować alfabetycznie listę dowolnej liczby słów lub nazw:

```
def alphabetize(original_list=[]):

""" Pass any list in square brackets, displays a string with items

sorted """

# Inside the function make a working copy of the list passed in.

sorted_list = original_list.copy()

# Sort the working copy.

sorted_list.sort()

# Make a new empty string for output

final_list = ''

# Loop through sorted list and append name and comma and space.

for name in sorted_list:

final_list = name ', '

# Knock of last comma space if final list is long enough

final_list = final_list[:-2]

# Print the alphabetized list.

print(final_list)
```

Przeprowadzimy Cię przez tę funkcję, aby wyjaśnić, co się dzieje. Pierwsza linia definiuje funkcję. Zauważysz, że dla parametru umieściliśmy original_list=[]. Wartość domyślna (=[]) jest opcjonalna, ale umieściliśmy ją tam z dwóch powodów. Po pierwsze, jeśli przypadkowo wywołasz go bez podania listy, nie „zawiesza się". Po prostu zwraca pustą listę. Docstring zawiera dodatkowe informacje. Na przykład, gdy zaczniesz wpisywać nazwę funkcji w VS Code, otrzymasz zarówno instrukcję def, jak i dokumentację, ponieważ funkcja IntelliSense przypomina o korzystaniu z funkcji, jak pokazano na rysunku.




Ponieważ funkcja nie może bezpośrednio zmienić listy, najpierw tworzy kopię oryginalnej listy (tej, która została przekazana) w nowej liście o nazwie sorted_list, z następującym wierszem kodu:

```
sorted_list = original_list.copy()
```

W tym momencie sorted_list nie jest tak naprawdę posortowana, nadal jest tylko kopią oryginału. Następna linia kodu dokonuje faktycznego sortowania:

```
sorted_list.sort()
```

Ta funkcja faktycznie tworzy ciąg z posortowanymi elementami oddzielonymi przecinkami. Tak więc ta linia tworzy nową nazwę zmiennej, final_list, i rozpoczyna ją jako pusty ciąg znaków po znaku = (to są dwa pojedyncze cudzysłowy bez spacji między nimi).

final_list = ''

Ta pętla przechodzi przez posortowaną listę i dodaje każdy element z listy, oddzielony przecinkiem i spacją, do łańcucha final_list:

for name in sorted_list:

final_list = name ', '

Gdy to zrobisz, jeśli w ogóle coś zostanie dodane do final_list, będzie miało dodatkowy przecinek i spację na końcu. Tak więc ta instrukcja usuwa te dwa ostatnie znaki, zakładając, że lista ma co najmniej dwa znaki:

final_list = final_list[:-2]

Następna instrukcja po prostu drukuje final_list, abyś mógł ją zobaczyć. Aby wywołać funkcję, możesz przekazać listę bezpośrednio w nawiasach funkcji, na przykład tak:

alphabetize(['Schrepfer', 'Maier', 'Santiago', 'Adams'])

Jak zawsze, możesz również przekazać nazwę zmiennej, która już zawiera listę, tak jak w tym przykładzie.

names = ['Schrepfer', 'Maier', 'Santiago', 'Adams']

alphabetize(names)

Tak czy inaczej, funkcja wyświetla te nazwy w kolejności alfabetycznej w następujący sposób.

Adams, Maier, Santiago, Schrepfer

**Przekazywanie dowolnej liczby argumentów**

Lista zapewnia jeden ze sposobów przekazywania wielu wartości do funkcji. Możesz także zaprojektować funkcję tak, aby akceptowała dowolną liczbę argumentów. Nie jest to szczególnie szybsze ani lepsze, więc nie ma „właściwego czasu" lub „niewłaściwego czasu" na użycie tej metody. Po prostu użyj tego, co wydaje ci się najłatwiejsze lub tego, co w danym momencie wydaje się najbardziej sensowne. Aby przekazać dowolną liczbę argumentów, użyj *args jako nazwy parametru, na przykład:

def sorter(*args)

Cokolwiek przekażesz, staje się krotką o nazwie args wewnątrz funkcji. Krotka to niezmienna lista (lista, której nie można zmienić). Więc znowu, jeśli chcesz coś zmienić, musisz skopiować krotkę do listy, a następnie pracować nad tą kopią. Oto przykład, w którym kod wykorzystuje prostą instrukcję newlist = list(args) (możesz to przeczytać, ponieważ zmienna o nazwie newlist jest listą wszystkich rzeczy, które znajdują się w krotce args). Następna linia, newlist.sort() sortuje listę, a print wyświetla zawartość listy:

def sorter(*args):

""" Pass in any number of arguments separated by commas

Inside the function, they treated as a tuple named args """

# The passed-in

# Create a list from the passed-in tuple

newlist = list(args)

# Sort and show the list.

newlist.sort()

print(newlist)

Rysunek przedstawia przykład uruchomienia tego kodu z serią liczb jako argumentami w komórce Jupytera. Jak widać, wynikowa lista jest posortowana zgodnie z oczekiwaniami.

```
def sorter(*args):
    """ Pass in any number of arguments separated by commas
    Inside the function, they treated as a tuple named args """
    # The passed-in
    # Create a list from the passed-in tuple
    newlist = list(args)
    # Sort and show the list.
    newlist.sort()
    print(newlist)

sorter(1, 0.001, 100000,-900,  2)
```

```
[-900, 0.001, 1, 2, 100000]
```

**Zwracanie wartości z funkcji**

Do tej pory wszystkie nasze funkcje po prostu wyświetlały dane wyjściowe na ekranie, dzięki czemu możesz upewnić się, że funkcja działa. W prawdziwym życiu częściej zdarza się, że funkcja zwraca pewną wartość i umieszcza ją w zmiennej określonej w kodzie wywołującym. Zazwyczaj jest to ostatni wiersz funkcji, po którym następuje spacja i nazwa zmiennej (lub jakiegoś wyrażenia), która zawiera wartość, która ma zostać zwrócona. Oto wariacja na temat funkcji alfabetycznej. Nie zawiera instrukcji drukowania. Zamiast tego na końcu po prostu zwraca final_list utworzoną przez funkcję:

def alphabetize(original_list=[]):

""" Pass any list in square brackets, displays a string with items

sorted """

# Inside the function make a working copy of the list passed in.

sorted_list = original_list.copy()

# Sort the working copy.

sorted_list.sort()

# Make a new empty string for output

final_list = ''

# Loop through sorted list and append name and comma and space.

for name in sorted_list:

final_list = name ', '

# Knock of last comma space if final list is long enough

final_list = final_list[:-2]

# Return the alphabetized list.

return final_list

Najczęstszym sposobem używania funkcji jest przechowywanie wszystkiego, co zwracają, w jakiejś zmiennej. Na przykład w poniższym kodzie pierwszy wiersz definiuje zmienną o nazwie random_list, która jest po prostu listą zawierającą nazwy w dowolnej kolejności, ujęte w nawiasy kwadratowe (co mówi Pythonowi, że to lista). Drugi wiersz tworzy nową zmienną o nazwie alpha_list, przekazując random_list do funkcji Alphabetize() i przechowując wszystko, co zwróci ta funkcja. Końcowa instrukcja print wyświetla wszystko, co znajduje się w zmiennej alpha_list:

random_list = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher',

'Jacobs']

alpha_list = alphabetize(random_list)

print(alpha_list)

Rysunek pokazuje wynik uruchomienia całego zestawu i kaboodla w komórce Jupytera

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock of last comma space if final list is long enough
    final_list = final_list[:-2]
    # Print the alphabetized list.
    print(final_list)

names = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher', 'Jacobs']
alphabetize(names)
```

```
Gallagher, Jacobs, Keaser, Maier, McMullen, Wilson, Yudt
```

**Demaskowanie funkcji anonimowych**

Python obsługuje koncepcję funkcji anonimowych, zwanych także funkcjami lambda. Anonimowa część nazwy opiera się na fakcie, że funkcja nie musi mieć nazwy (ale może ją mieć, jeśli chcesz). Część lambda opiera się na użyciu słowa kluczowego lambda do zdefiniowania ich w Pythonie. Lambda jest także 11. literą alfabetu greckiego. Ale głównym powodem, dla którego ta nazwa jest używana w Pythonie, jest to, że termin lambda jest używany do opisywania anonimowych funkcji w rachunku różniczkowym. Teraz, gdy to wyjaśniliśmy, możesz wykorzystać te informacje do wywołania fascynującej rozmowy na imprezach biurowych. Minimalna składnia definiowania wyrażenia lambda (bez nazwy) to:

Lambda arguments : expression

Podczas korzystania z niego:

* Zastąp argumenty danymi przekazywanymi do wyrażenia.

* Zastąp wyrażenie wyrażeniem (formułą), które określa, co ma zwracać lambda

Dość powszechnym przykładem użycia tej składni jest próba sortowania ciągów tekstowych, w których niektóre nazwy zaczynają się od wielkich liter, a inne od małych liter, jak w tych nazwach:

Adams, Ma, DiMeola, Zandusky

Załóżmy, że piszesz następujący kod, aby umieścić nazwy na liście, posortować ją, a następnie wydrukować listę, tak jak poniżej:

names = ['Adams', 'Ma', 'diMeola', 'Zandusky']

names.sort()

print(names)

To wyjście z tego to:

['Adams', 'Ma', 'Zandusky', 'diMeola']

To, że DiMeola przyszedł po Zandusky'ego, wydaje się niewłaściwe dla mnie i prawdopodobnie dla ciebie. Ale komputery nie zawsze widzą rzeczy tak, jak my. (Właściwie nic nie „widzą”, ponieważ nie mają oczu ani mózgu… ale to nie ma znaczenia.) Powodem, dla którego diMeola pojawia się po Zandusky, jest to, że sortowanie jest oparte na ASCII, który jest systemem w w którym każdy znak jest reprezentowany przez liczbę. Wszystkie małe litery mają cyfry, które są większe niż wielkie cyfry. Tak więc podczas sortowania wszystkie słowa zaczynające się od małych liter występują po słowach rozpoczynających się od dużej litery. Jeśli nic innego, to przynajmniej gwarantuje drobne hmm. Aby pomóc w tych kwestiach, metoda Pythona sort() pozwala umieścić wyrażenie key= w nawiasach, gdzie można określić sposób sortowania. Składnia to:

.sort(key = transform)

Część przekształcająca to pewna odmiana sortowanych danych. Jeśli masz szczęście i jedna z wbudowanych funkcji, takich jak len (dla długości), będzie działać dla Ciebie, możesz po prostu użyć jej zamiast transformacji, tak jak poniżej:

names.sort(key=len)

Niestety dla nas długość sznurka nie pomaga w ułożeniu alfabetycznym. Więc kiedy to uruchomisz, kolejność okazuje się następująca:

['Ma', 'Adams', 'diMeola', 'Zandusky']

Sortowanie przebiega od najkrótszego łańcucha (tego z najmniejszą liczbą znaków) do najdłuższego ciągu. W tej chwili nie jest pomocny.

Nie możesz napisać key=lower lub key=upper, aby oprzeć sortowanie na wszystkich małych lub wszystkich wielkich literach, ponieważ dolna i górna nie są wbudowanymi funkcjami (które możesz dość szybko zweryfikować, przeglądając wbudowane w python 3.7 Funkcje). Zamiast funkcji wbudowanej możesz użyć funkcji niestandardowej, którą sam zdefiniujesz za pomocą def. Na przykład możemy utworzyć funkcję o nazwie lower(), która akceptuje

ciąg i zwraca ten ciąg ze wszystkimi jego literami przekonwertowanymi na małe litery. Oto funkcja:

def lower(anystring):

""" Converts string to all lowercase """

return anystring.lower()

Nazwa lower to nazwa, którą wymyśliłem, a anystring jest symbolem zastępczym dowolnego ciągu, który przekażesz do niej w przyszłości. Funkcja return anystring.lower() zwraca ciąg przekonwertowany na małe litery przy użyciu metody .lower() obiektu str (string). Nie możesz użyć key=lower w nawiasach sort(), ponieważ lower() nie jest funkcją wbudowaną. To metoda. . . nie ten sam. Wiem, trochę irytujące z tymi wszystkimi modnymi hasłami. Załóżmy, że piszesz tę funkcję w komórce Jupytera lub pliku .py. Następnie wywołujesz funkcję z czymś takim jak print(lowercaseof('Zandusky')). To, co otrzymasz jako wynik, to łańcuch przekonwertowany na małe litery, jak na rysunku.


Okej, więc teraz mamy niestandardową funkcję konwertującą dowolny ciąg na wszystkie małe litery. Jak możemy użyć tego jako klucza sortowania? Łatwo, użyj key=transform tak samo jak poprzednio, ale zamień transform na niestandardową nazwę funkcji. Nasza funkcja nosi nazwę lowercaseof, więc użylibyśmy .sort(key=lowcaseof), jak pokazano poniżej:

def lowercaseof(anystring):

""" Converts string to all lowercase """

return anystring.lower()

names = ['Adams', 'Ma', 'diMeola', 'Zandusky']

names.sort(key=lowercaseof)

Uruchomienie tego kodu w celu wyświetlenia listy nazw ustawia je we właściwej kolejności, ponieważ sortowanie opiera się na ciągach, które wszystkie są pisane małymi literami. Wynik jest taki sam jak poprzednio, ponieważ tylko sortowanie, które miało miejsce za kulisami, używało małych liter. Oryginalne dane są nadal zapisane wielkimi i małymi literami.

„Adams", „diMeola", „Ma", „Zandusky"

Jeśli po przeczytaniu tego wszystkiego nadal nie śpisz i jesteś świadomy, możesz pomyśleć: „Dobrze, rozwiązałeś problem z sortowaniem. Ale myślałem, że mówimy tutaj o funkcjach lambda. Gdzie jest funkcja lambda?" Nie ma jeszcze funkcji lambda. Jest to jednak doskonały przykład zastosowania funkcji lambda, ponieważ funkcja, którą wywołujesz, lowercaseof(), wykonuje całą swoją pracę za pomocą jednego wiersza` kodu: zwróć anystring.lower(). Kiedy twoja funkcja może wykonać swoje zadanie za pomocą takiego prostego, jednowierszowego wyrażenia, możesz pominąć def i nazwę funkcji i po prostu użyć następującej składni:

lambda parameters : expression

Zamień parametry na jedną lub więcej nazw parametrów, które sam wymyślisz (nazwy w nawiasach po def i nazwa funkcji w zwykłej funkcji). Zamień wyrażenie na to, co chcesz, aby funkcja zwróciła bez zwracania słowa. Tak więc w tym przykładzie kluczem, używając wyrażenia lambda, byłoby:

lambda anystring : anystring.lower()

Teraz możesz zobaczyć, dlaczego jest to funkcja anonimowa. Usunięto całą pierwszą linię z nazwą funkcji smallcaseof(). Tak więc zaletą używania wyrażenia lambda jest to, że w ogóle nie potrzebujesz zewnętrznej funkcji niestandardowej. Potrzebujesz tylko parametru, po którym następuje dwukropek i wyrażenie, które mówi mu, co zwrócić. Rysunek 5-12 przedstawia pełny kod i wynik jego uruchomienia.

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key = lambda anystring : anystring.lower())
print(names)


['Adams', 'diMeola', 'Ma', 'Zandusky']
```

Otrzymujesz właściwy porządek sortowania bez potrzeby stosowania zewnętrznej funkcji klienta, takiej jak smallcaseof(). Wystarczy użyć anystring : anystring.lower() (po słowie lambda) jako klucza sortowania.

Mogę również dodać, że anystring to dłuższa nazwa parametru, niż użyłaby większość Pythonistów. Ludzie w Pythonie lubią krótkie nazwy, nawet jednoliterowe. Na przykład możesz zamienić dowolny ciąg na s (lub dowolną inną literę), jak poniżej, a kod będzie działał dokładnie tak samo:

names = ['Adams', 'Ma', 'diMeola', 'Zandusky']

names.sort(key=lambda s: s.lower())

print(names)

Już na początku tej tyrady wspomniałem, że funkcja lambda nie musi być anonimowa. Możesz nadać im nazwy i wywoływać je tak, jak inne funkcje. Oto na przykład funkcja lambda o nazwie waluta, która pobiera dowolną liczbę i zwraca ciąg znaków w formacie walutowym (to znaczy ze znakiem dolara na początku, przecinkami między tysiącami i dwiema cyframi oznaczającymi grosze):

currency = lambda n : f"${n:,.2f}"

Oto jeden nazwany procent, który mnoży dowolną liczbę, którą do niego wysyłasz, przez 100 i pokazuje ją z dwoma znakami procenta na końcu:

percent = lambda n : f"{n:.2%}"

Rysunek  przedstawia przykłady obu funkcji zdefiniowanych na górze komórki Jupytera. następnie kilka instrukcji print wywołuje funkcje po imieniu i przekazuje im przykładowe dane.

```
# Show number in currency format.
currency = lambda n : f"${n:,.2f}"
# Show number in percent format.
percent = lambda n : f"{n:.2%}"

# Test currency function
print(currency(99))
print(currency(123456789.09876543))

# Test percent function
print(percent(0.065))
print(percent(.5))
```

```
$99.00
$123,456,789.10
6.50%
50.00%
```

Każda instrukcja print() wyświetla liczbę w żądanym formacie. Powodem, dla którego możesz zdefiniować je jako lambdy jednowierszowe, jest to, że możesz zrobić całą pracę w jednym wierszu, f"${n:,.2f}" dla pierwszego i f"{n:.2%}" dla drugiego. Ale to, że możesz to zrobić w ten sposób, nie oznacza, że musisz. Możesz także użyć zwykłych funkcji, jak następuje:

# Show number in currency format.

def currency(n):

return f"${n:,.2f}"

def percent(n):

# Show number in percent format.

return f"{n:.2%}"

Dzięki tej dłuższej składni możesz również przekazać więcej informacji. Na przykład możesz domyślnie ustawić format wyrównany do prawej w ramach określonej szerokości (powiedzmy 15 znaków), aby wszystkie liczby były wyrównane do prawej do tej samej szerokości. Rysunek pokazuje tę odmianę dwóch funkcji.

```
# Show number in currency format, specify width.
def currency(n, w=15):
    """ Show in currency format, width = 15 or width of your choosing """
    s = f"${n:,.2f}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)

# Show number in percent format, specify width.
def percent(n, w=15):
    """ Show in percent format, width = 15 or width of your choosing """
    # Show number in percent format.
    s = f"{n:.1%}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)
```

Na rysunku drugi parametr jest opcjonalny iw przypadku pominięcia przyjmuje wartość domyślną 15. Więc jeśli nazwiesz to tak:

print(currency(9999))

. . . otrzymujesz dopełnienie w wysokości 9 999,00 $ z wystarczającą ilością spacji po lewej stronie, aby miało szerokość 15 znaków. Jeśli zamiast tego nazwiesz to tak:

print(currency(9999,20)

. . . nadal dostajesz 9 999,00 $, ale uzupełnione wystarczającą ilością spacji po lewej stronie, aby mieć szerokość 20 znaków.

Metoda .ljust() użyta na rysunku 5.14 jest wbudowaną w Pythona metodą łańcuchową, która wypełnia lewą stronę łańcucha wystarczającą ilością spacji, aby nadać mu określoną szerokość. Istnieje również metoda rjust() do wypełniania prawej strony. Można również określić znak inny niż spacja. Google python 3 ljust rjust, jeśli potrzebujesz więcej informacji. Więc masz to, możliwość tworzenia własnych niestandardowych funkcji w Pythonie. W prawdziwym życiu to, co chcesz zrobić, to za każdym razem, gdy okaże się, że potrzebujesz dostępu do tego samego fragmentu kodu — tego samego fragmentu logowania — w swojej aplikacji w kółko, nie po prostu kopiuj/wklej ten fragment kod w kółko. Zamiast tego umieść cały ten kod w funkcji, którą możesz wywołać po imieniu. W ten sposób, jeśli zdecydujesz się zmienić kod, nie musisz przekopywać się przez aplikację, aby znaleźć wszystkie miejsca, które wymagają zmiany. Po prostu zmień to w funkcji, gdzie wszystko jest zdefiniowane w jednym miejscu.