Rozdział 7

Pliki

7.1. Pamięć nieulotna

Do tej pory uczyliśmy się pisać programy i komunikować *procesorowi* nasze zamiary za pomocą instrukcji warunkowych, funkcji i iteracji. Nauczyliśmy się, jak tworzyć i wykorzystywać struktury danych w *pamięci głównej*. Procesor i pamięć główna są miejscem, w którym działa i uruchamia się nasze oprogramowanie. To właśnie tam odbywa się całe „myślenie".

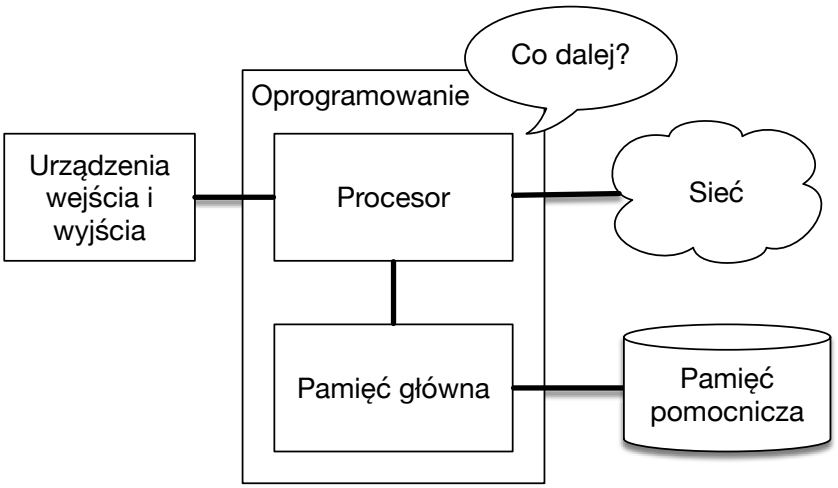
Przypomnij sobie naszą dyskusję na temat architektury sprzętowej. Po wyłączeniu zasilania wszystko, co jest zapisane na procesorze lub w pamięci głównej, zostanie usunięte. Tak więc do tej pory nasze programy były tylko przejściowymi, zabawowymi ćwiczeniami do nauki Pythona.

W tym rozdziale rozpoczynamy pracę z *pamięcią pomocniczą* (lub plikami). Pamięć pomocnicza nie jest czyszczona po wyłączeniu zasilania. W przypadku pendrive'a dane, które zapisujemy z naszych programów, mogą zostać usunięte z systemu i przeniesione do innego systemu.

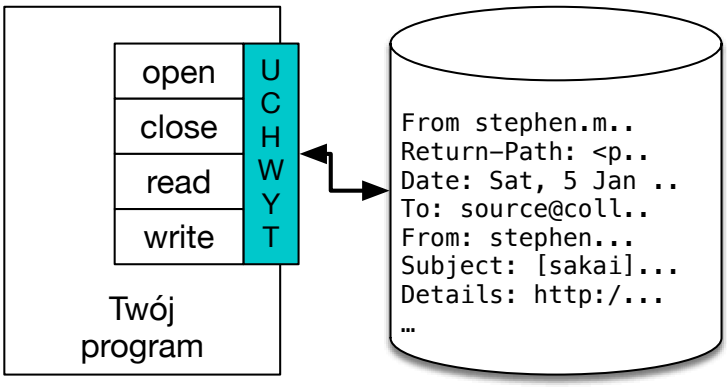
Skupimy się przede wszystkim na odczycie i zapisie plików tekstowych, takich jak te, które tworzymy w edytorze tekstu. Później zobaczymy, jak pracować z plikami bazodanowymi, które są plikami binarnymi, przeznaczonymi specjalnie do odczytu i zapisu przez oprogramowanie bazodanowe.

7.2. Otwieranie plików

Kiedy chcemy odczytać lub zapisać plik (np. na dysku twardym), musimy najpierw *otworzyć* plik. Otwarcie pliku komunikuje się z Twoim systemem operacyjnym, który wie, gdzie przechowywane są dane dla każdego pliku.



Rysunek 7.1. Pamięć pomocnicza



Rysunek 7.2. Uchwyt pliku

Gdy otwierasz plik, prosisz system operacyjny o znalezienie go po nazwie i o upewnienie się, że ten plik istnieje. W poniższym przykładzie otwieramy plik `mbox.txt`, który powinien być przechowywany w tym samym katalogu, w którym znajdujesz się po uruchomieniu Pythona. Możesz pobrać ten plik z https://py4e.pl/code3/mbox.txt.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Jeśli funkcja `open()` się powiedzie, system operacyjny zwróci nam *uchwyt pliku*. Uchwyt pliku nie jest rzeczywistymi danymi zawartymi w tym pliku, lecz „uchwytem", który możemy użyć do odczytania danych. Otrzymujesz uchwyt, jeśli żądany plik istnieje i masz odpowiednie uprawnienia do jego odczytania.

Jeśli plik nie istnieje, `open()` się nie powiedzie, a Ty nie będziesz mógł uzyskać dostępu do jego zawartości:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Później użyjemy `try` i `except`, tak aby zgrabniej poradzić sobie z sytuacją, w której próbujemy otworzyć nieistniejący plik.

7.3. Pliki tekstowe i linie

Plik tekstowy może być uważany za sekwencję linii, podobnie jak ciąg znaków w Pythonie może być uważany za sekwencję liter, liczb i symboli. Na przykład poniżej znajduje się fragment pliku tekstowego, który rejestruje aktywność mailową różnych osób w zespole rozwijającym projekt otwartego oprogramowania:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Cały plik interakcji mailowych jest dostępny pod adresem:

https://py4e.pl/code3/mbox.txt

a skrócona wersja pliku, pod adresem:

https://py4e.pl/code3/mbox-short.txt

Pliki te są w standardowym formacie pliku tekstowego, który zawiera wiele wiadomości e-mail. Wiersze rozpoczynające się od „From" oddzielają wiadomości, a wiersze rozpoczynające się od „From:" są częścią wiadomości. Więcej informacji na temat formatu Mbox można znaleźć na stronie https://pl.wikipedia.org/wiki/Mbox.

Aby podzielić plik na linie, istnieje specjalny znak, który reprezentuje „koniec linii", zwany nomen omen znakiem *końca linii*.

W ciągach znaków Pythona znak *końca linii* reprezentujemy jako lewy ukośnik-n. Nawet jeśli wygląda to jak dwa znaki, to w rzeczywistości jest to pojedynczy znak. W poniższym przykładzie, gdy patrzymy na zmienną `stuff`, po wpisaniu jej w interpreterze, pokazuje nam ona `\n` w ciągu znaków, ale gdy używamy `print()` do wyświetlenia ciągu znaków, widzimy go rozbitego na dwie linie przez znak końca linii.

```
>>> stuff = 'Witaj\nświecie!'
>>> stuff
'Witaj\nświecie!'
>>> print(stuff)
Witaj
świecie!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

Widzimy też, że długość ciągu znaków `X\nY` to *trzy* znaki, ponieważ znak końca linii jest pojedynczym znakiem.

Kiedy więc patrzymy na linie w pliku, musimy sobie *wyobrazić*, że na końcu każdej linii znajduje się specjalny niewidoczny znak zwany znakiem końca linii.

Tak więc znak końca linii dzieli znaki w pliku na linie.

7.4. Czytanie plików

Podczas gdy *uchwyt pliku* nie zawiera danych pliku, to całkiem łatwo jest skonstruować pętlę `for` do jego odczytu i zliczyć linie występujące w pliku:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Liczba linii:', count)

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

Możemy użyć uchwytu pliku jako sekwencji w naszej pętli `for`. Nasza pętla `for` po prostu zlicza linie w pliku i wyświetla sumę. Tłumaczenie treści pętli `for` na polski jest z grubsza następujące: „dla każdej linii w pliku reprezentowanej przez uchwyt pliku, dodaj jeden do zmiennej `count`".

Powodem, dla którego funkcja `open()` nie odczytuje całego pliku jest to, że plik może być dość duży i zawierać wiele gigabajtów danych. Funkcja `open()` zajmuje zawsze tyle samo czasu, niezależnie od wielkości pliku. Pętla `for` w rzeczywistości powoduje, że dane są odczytywane z pliku.

Gdy plik jest w ten sposób odczytywany przy użyciu pętli `for`, Python zajmuje się podziałem danych z pliku na osobne linie przy użyciu znaku końca linii. Python w każdej iteracji pętli `for` czyta każdą linię od początku aż do wystąpienia znaku końca linii i dołącza go jako ostatni znak w zmiennej `line`.

Ponieważ pętla `for` odczytuje z danych jeden wiersz na raz, może efektywnie czytać i zliczać wiersze w bardzo dużych plikach, bez obawy, że w pamięci głównej skończy się miejsce na dane. Powyższy program przy użyciu bardzo małej ilości pamięci może zliczać linie w plikach o dowolnym rozmiarze, ponieważ każda linia jest odczytywana, zliczana, a następnie porzucana.

Jeśli wiesz, że plik jest stosunkowo mały w porównaniu z rozmiarem Twojej pamięci głównej, możesz odczytać cały plik za jednym zamachem, używając metody `read()` na uchwycie pliku.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

W powyższym przykładzie cała zawartość (wszystkie 94 626 znaków) pliku `mbox-short.txt` jest wczytywana bezpośrednio do zmiennej `inp`. Do wyświetlenia pierwszych 20 znaków danych z ciągu znaków przechowywanych w `inp` używamy operacji wycinana podciągów.

Gdy plik zostanie odczytany w ten sposób, to wszystkie znaki, łącznie ze wszystkimi liniami i znakami końca linii, są jednym dużym ciągiem znaków znajdującym się w zmiennej `inp`. Dobrym pomysłem jest przechowywanie wyniku funkcji `read()` jako zmiennej, ponieważ każde wywołanie `read()` wykorzystuje zasoby pamięci:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

Pamiętaj, że ta forma funkcji `open()` powinna być używana tylko wtedy, gdy dane pliku zmieszczą się spokojnie w pamięci głównej komputera. Jeśli plik jest zbyt duży, by zmieścić się w pamięci głównej, powinieneś napisać swój program tak, aby odczytywać plik po kawałku, używając pętli `for` lub `while`.

7.5. Przeszukiwanie pliku

Kiedy przeszukujesz dane w pliku, bardzo powszechnym sposobem jest odczytywanie ich, ignorując przy tym większość linii i przetwarzając tylko te, które spełniają określony warunek – patrząc na to ogólniej, nazywamy to *schematem filtrowania*. Możemy połączyć przebieg odczytywania pliku z metodami ciągów znaków, tak aby zbudować proste mechanizmy wyszukiwania.

Na przykład, jeśli chcielibyśmy odczytać plik i wypisać tylko te linie, które rozpoczęły się od „From:", moglibyśmy użyć metody ciągów znaków `startswith()`, tak aby wybrać tylko te, które mają pożądany początek:

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/search1.py
```

Gdy powyższy program zostanie uruchomiony, otrzymamy następujący wynik:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu
```

```
From: rjlowe@iupui.edu
...
```

Wynik wygląda świetnie, ponieważ jedyne linie, które widzimy, to te, które zaczynają się od „From:", ale dlaczego widzimy dodatkowe puste linie? Jest to spowodowane wspomnianym wcześniej niewidocznym *znakiem końca linii*. Każda z linii kończy się wspomnianym znakiem, więc funkcja `print()` wypisuje ciąg znaków zapisany w zmiennej `line`, który zawiera znak końca linii, a następnie dodaje *kolejny* znak końca linii, co daje efekt podwójnego odstępu, który widzimy.

Moglibyśmy użyć wycinka linii, aby wypisać wszystkie znaki oprócz ostatniego, ale prostszym podejściem jest użycie metody `rstrip()`, która usuwa spacje z prawej strony ciągu znaków:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/search2.py
```

Po uruchomieniu tego programu otrzymamy następujący wynik:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

W miarę jak Twoje programy do przetwarzania plików będą stawać się coraz bardziej skomplikowane, możesz chcieć uporządkować swoje pętle wyszukiwania za pomocą `continue`. Podstawową ideą pętli wyszukiwania jest to, że szukasz „interesujących" linii i skutecznie pomijasz „nieciekawe". A potem, gdy znajdziemy interesującą linię, coś z nią robimy.

Pętlę możemy uporządkować tak, aby w następujący sposób trzymała się schematu pomijania „nieciekawych" linii:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Pomiń 'nieciekawe' linie
    if not line.startswith('From:'):
        continue
    # Przetwarzaj 'interesujące' linie
    print(line)

# Kod źródłowy: https://py4e.pl/code3/search3.py
```

Wynik programu jest taki sam. Mówiąc po ludzku, „nieciekawe" są te linie, które nie rozpoczynają się od „From:", więc pomijamy je, używając `continue`. W przypadku „interesujących" linii (tzn. tych, które zaczynają się od „From:") wykonujemy przetwarzanie danych, czyli wypisujemy ich zawartość na ekran.

Możemy użyć metody ciągów znaków `find()`, tak aby zasymulować wyszukiwanie w edytorze tekstu znajdujące linie, w których poszukiwany tekst jest gdziekolwiek w linii. Ponieważ `find()` szuka wystąpienia ciągu znaków w innym ciągu i albo zwraca jego pozycję, albo -1, jeśli nie zostanie on znaleziony, możemy napisać następującą pętlę, tak by pokazać te linie, które zawierają ciąg znaków „@uct.ac.za" (tzn. pochodzą z Uniwersytetu w Kapsztadzie w RPA):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)
```

 # Kod źródłowy: https://py4e.pl/code3/search4.py

Otrzymujemy następujący wynik:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Używamy tutaj skróconej formy instrukcji `if` – umieściliśmy `continue` w tej samej linii co `if`. Ta skrócona forma `if` funkcjonuje tak samo, jak gdyby `continue` znajdowało się w następnej linii i było wcięte.

7.6. Pozwolenie użytkownikowi na wybranie nazwy pliku

Naprawdę nie chcemy edytować naszego kodu Pythona za każdym razem, gdy będziemy przetwarzać nowy plik. Bardziej użyteczne byłoby poproszenie użytkownika o wprowadzenie nazwy pliku za każdym razem, gdy program zostanie uruchomiony, tak aby mógł on korzystać z naszego programu na różnych plikach, bez konieczności zmiany kodu Pythona.

Jest to dość proste do zrobienia poprzez odczytanie nazwy pliku od użytkownika za pomocą `input()` w następujący sposób:

```
fname = input('Podaj nazwę pliku: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('Mamy', count, 'linii z tematem wiadomości w pliku', fname)
```

 # Kod źródłowy: https://py4e.pl/code3/search6.py

Odczytujemy nazwę pliku wprowadzoną przez użytkownika, umieszczamy ją w zmiennej o nazwie `fname` i otwieramy ten plik. Teraz możemy uruchamiać program wielokrotnie na różnych plikach.

```
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt
```

```
Podaj nazwę pliku: mbox-short.txt
Mamy 27 linii z tematem wiadomości w pliku mbox-short.txt
```

Zanim przejdziemy do następnej sekcji, spójrzmy na powyższy program i zadajmy sobie pytanie: „Co tu się może nie udać?" lub „Co mógłby zrobić nasz miły użytkownik, żeby nasz mały program zakończył się błędem z komunikatem mechanizmu *traceback*, co sprawiłoby, że wyjdziemy na kiepskich programistów?".

7.7. Używanie `try`, `except` i `open()`

Mówiłem Ci, żebyś nie podglądał. To Twoja ostatnia szansa.

A co, jeśli nasz użytkownik wpisze coś, co nie jest nazwą pliku?

```
Podaj nazwę pliku: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

Podaj nazwę pliku: szurum-burum
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'szurum-burum'
```

Nie śmiej się. Użytkownicy czasem robią wszystko, co w ich mocy, by wysypać Twoje programy – albo przypadkiem, albo w złych zamiarach. W rzeczywistości ważną częścią każdego zespołu zajmującego się tworzeniem oprogramowania jest osoba lub grupa o nazwie *Quality Assurance* (w skrócie QA; zespół *zapewniania jakości*), której zadaniem jest robienie najbardziej szalonych rzeczy w celu wysypania oprogramowania stworzonego przez programistów.

Zespół QA jest odpowiedzialny za znalezienie wad w programach, zanim dostarczymy je użytkownikom końcowym, którzy mogą je zakupić lub zapłacić nam za napisanie programu. Tak więc zespół QA jest najlepszym przyjacielem programisty.

Więc teraz, gdy widzimy wadę w programie, możemy ją elegancko naprawić, używając struktury `try/except`. Musimy przyjąć, że `open()` może się nie powieść, więc dodamy kod naprawczy, który zadziała, gdy tak się stanie:

```
fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('Mamy', count, 'linii z tematem wiadomości w pliku', fname)

# Kod źródłowy: https://py4e.pl/code3/search7.py
```

Funkcja `exit()` kończy program. Jest to funkcja, którą wywołujemy, ale która nigdy nie wraca do miejsca wywołania. Teraz, gdy nasz użytkownik (lub zespół QA) wpisze jakieś głupie lub złe nazwy plików, „łapiemy" je i elegancko przywracamy normalne działanie:

```
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt

Podaj nazwę pliku: szurum-burum
Nie można otworzyć pliku: szurum-burum
```

Ochrona wywołania `open()` jest dobrym przykładem prawidłowego użycia `try` i `except` w programie Pythona. W języku angielskim używamy terminu *pythonic* („pythonowy") wtedy, gdy robimy coś „w stylu Pythona". Możemy powiedzieć, że powyższy przykład jest „pythonowym" sposobem na otwarcie pliku.

Kiedy będziesz bardziej biegły w Pythonie, będziesz mógł brać udział w błyskotliwej wymianie zdań z innymi programistami Pythona, tak aby zdecydować, które z dwóch równoważnych rozwiązań problemu jest „bardziej pythonowe". Bycie „bardziej pythonowym" wynika z idei, że programowanie jest zarówno inżynierią, jak i sztuką. Nie zawsze jesteśmy zainteresowani tym, aby coś po prostu działało; chcemy również, by nasze rozwiązanie było eleganckie, co zauważą i docenią równi nam programiści.

7.8. Pisanie do plików

Aby móc zapisywać dane do pliku, musisz otworzyć go w trybie `w` (skrót z ang. *write*), podanym jako drugi argument:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Jeśli plik już istnieje, otwarcie go w trybie do zapisu usuwa stare dane i tworzy nowy pusty plik, więc bądź ostrożny! Jeśli plik nie istnieje, tworzony jest nowy.

Metoda `write()` obiektu uchwytu pliku umieszcza dane w pliku, zwracając liczbę zapisanych znaków. Domyślnym trybem zapisu jest tryb tekstowy, przeznaczony do zapisu (i odczytu) ciągów znaków. Dodajmy jedną linię do naszego pliku (nie używaj polskich znaków – wrócimy do tego problemu za chwilę):

```
>>> line1 = "Oto galaz eukaliptusa,\n"
>>> fout.write(line1)
23
```

Obiekt pliku śledzi, gdzie się obecnie znajduje, więc jeśli ponownie wywołasz metodę `write()`, to nowe dane zostaną umieszczone na końcu pliku.

Podczas zapisu do pliku musimy być pewni, że ogarniamy znaki końca linii – tutaj wyraźnie wstawiamy znak końca linii, gdy tę linię chcemy zakończyć. Funkcja `print()` automatycznie dołącza znak nowej linii, ale metoda `write()` – nie.

```
>>> line2 = 'ojczyzny naszej emblemat.\n'
>>> fout.write(line2)
26
```

Kiedy skończysz pisać, musisz zamknąć plik, aby upewnić się, że ostatni bit danych jest fizycznie zapisany na dysku i aby nie został utracony w przypadku wyłączenia zasilania komputera.

```
>>> fout.close()
```

Możemy zamknąć również pliki, które otwieramy do odczytu, ale pozwolić sobie na odrobinę nieuwagi, jeśli otwieramy tylko kilka plików. Python sprawi, że po zakończeniu programu wszystkie otwarte pliki zostaną zamknięte. Kiedy zapisujemy dane do plików, lepiej osobiście zadbać o ich zamknięcie, tak aby niczego nie pozostawić przypadkowi.

7.9. Kodowanie plików

Przechowując dane tekstowe w plikach, należy określić, ile bitów potrzeba na zapisanie jednego znaku. Przez ostatnie lata zostało wypracowanych wiele systemów kodowania, np. klasyczny siedmiobitowy ASCII[1], obecnie

[1] https://pl.wikipedia.org/wiki/ASCII

będący standardem UTF-8[2], popularny w naszej części Europy ISO-8859-2[3] (znany również jako Latin-2) oraz CP-1250 (znany również jako Windows-1250[4]). Często informacja o kodowaniu pliku nie jest dostępna, a niektóre programy przetwarzające dane z góry zakładają jakieś kodowanie (np. UTF-8). W takich sytuacjach przetwarzanie pliku wejściowego w niepoprawnym kodowaniu da nam złe wyniki. W konsekwencji, pewną trudność może sprawić obsługa znaków diakrytycznych podczas odczytu i zapisu do pliku.

Dla przykładu użyjemy pliku `polski.txt`, który jest do ściągnięcia z poniższej strony:

https://py4e.pl/code3/polski.txt

Standardowo możemy otworzyć plik bez podawania żadnych argumentów:

```
>>> fhand = open('polski.txt')
```

W zależności od systemu operacyjnego możemy zauważyć, że wydrukowanie informacji o zmiennej `fhand` nieco się różni w parametrze `encoding`. Np. w angielskiej wersji systemu Windows możemy otrzymać informację o domyślnym kodowaniu CP-1252:

```
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='cp1252'>
```

Z kolei w systemie Linux domyślne kodowanie może być ustawione na UTF-8:

```
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='UTF-8'>
```

Jeśli w takiej sytuacji spróbujemy odczytać i wyświetlić zawartość pliku, to w przypadku powyższej wersji Windowsa otrzymamy tzw. krzaki:

```
>>> print(fhand.read())
Nikt nie spodziewa siÄ HiszpaÅ„skiej Inkwizycji.
WÅ>rÃ³d naszych metod sÄ... tak rÃ³Å¼ne elementy
jak strach, zaskoczenie... bezwzglÄdna skutecznoÅ>Ä‡
i niemalÅ¼e fanatyczne oddanie papieÅ¼owi...
oraz piÄkne czerwone mundurki.
```

W przypadku Linuxa otrzymamy prawidłowy tekst:

```
>>> print(fhand.read())
Nikt nie spodziewa się Hiszpańskiej Inkwizycji.
Wśród naszych metod są tak różne elementy
jak strach, zaskoczenie... bezwzględna skuteczność
i niemalże fanatyczne oddanie papieżowi...
oraz piękne czerwone mundurki.
```

Plik `polski.txt` jest zapisany w kodowaniu UTF-8. To, w jakim kodowaniu Python domyślnie otworzy plik, zależy od systemu operacyjnego. Na szczęście funkcja `open()` obsługuje parametr o nazwie `encoding`, w którym możemy podać, w jakim kodowaniu ma zostać otwarty plik. W przypadku Windowsa moglibyśmy otworzyć plik w następujący sposób:

```
>>> fhand = open('polski.txt', encoding='UTF-8')
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='UTF-8'>
>>> print(fhand.read())
```

[2] https://pl.wikipedia.org/wiki/UTF-8
[3] https://pl.wikipedia.org/wiki/ISO_8859-2
[4] https://pl.wikipedia.org/wiki/CP-1250

```
Nikt nie spodziewa się Hiszpańskiej Inkwizycji.
Wśród naszych metod są tak różne elementy
jak strach, zaskoczenie... bezwzględna skuteczność
i niemalże fanatyczne oddanie papieżowi...
oraz piękne czerwone mundurki.
```

Próbując zapisać tekst „żółć" pod Windowsem w domyślnym kodowaniu, prawdopodobnie trafimy na błąd kodowania[5]:

```
>>> fhand_pl1 = open('proba1.txt', 'w')
>>> print(fhand_pl1)
<_io.TextIOWrapper name='proba1.txt' mode='w' encoding='cp1252'>
>>> fhand_pl1.write("żółć")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python3\lib\encodings\cp1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u017c'
                    in position 0: character maps to <undefined>
>>> fhand_pl1.close()
```

Na szczęście możemy użyć parametru `encoding`:

```
>>> fhand_pl2 = open('proba2.txt', 'w', encoding='UTF-8')
>>> fhand_pl2.write("żółć")
4
>>> fhand_pl2.close()
```

Problemy z kodowaniem to szeroki temat i początkującemu programiście może to napsuć wiele krwi. Istnieje wiele sposobów na automatyczne wykrywanie kodowania i konwersji z jednego na drugie. Jednak na potrzeby tego kursu wystarczy, że w razie wątpliwości otworzysz plik np. w programistycznym edytorze tekstu Atom lub edytorze tekstu Notepad++ (informacja o kodowaniu pliku będzie w prawym dolnym rogu ekranu).

Obecnie ogólnie przyjętym standardem kodowania plików jest UTF-8.

7.10. Debugowanie

Kiedy odczytujesz dane lub zapisujesz dane do plików, możesz mieć problemy z białymi znakami. Tego typu błędy mogą być trudne do debugowania, ponieważ spacje, tabulacje i znaki końca linii są zazwyczaj niewidoczne:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

W trybie interaktywnym możemy wyświetlić zawartość takiej zmiennej po prostu poprzez wpisanie jej nazwy:

```
>>> s
'1 2\t 3\n 4'
```

W przypadku skryptu możemy skorzystać z wbudowanej funkcji `repr()`. Przyjmuje ona dowolny obiekt jako argument i zwraca jego reprezentację w postaci ciągu znaków. Jeśli obiektem jest właśnie ciąg znaków, to białe znaki są przedstawiane w postaci z lewym ukośnikiem:

[5] W polskiej wersji systemu Windows domyślnym kodowaniem zapewne będzie CP-1250 i próba zapisu polskiego tekstu zakończy się powodzeniem.

```
print(repr(s))
```

```
'1 2\t 3\n 4'
```

Innym problemem, na który możesz się natknąć, jest to, że różne systemy używają różnych znaków, by oznaczyć koniec linii. Niektóre systemy (np. Linux i macOS) używają znaku końca linii, reprezentowanego przez `\n`. Inne używają zamiast niego znaku powrotu karetki, reprezentowanego przez `\r`. Jeszcze inne (np. Windows) używają obu tych znaków, tj. `\r\n`. Jeśli przenosisz pliki między różnymi systemami, te niespójności mogą powodować problemy.[6]

Więcej o tym problemie możesz poczytać na stronie https://pl.wikipedia.org/wiki/Koniec_linii. Dla większości systemów istnieją aplikacje do konwersji z jednego formatu na drugi. Możesz je znaleźć na stronie https://en.wikipedia.org/wiki/Newline w sekcji „Conversion between newline formats". Oczywiście możesz też samodzielnie napisać taki konwerter.

7.11. Słowniczek

krzaki Niepoprawnie wyświetlane znaki tekstowe.
łapanie wyjątku Uniemożliwienie wyjątkowi zakończenia programu przy użyciu `try` i `except`.
plik tekstowy Sekwencja znaków zapisana na nośniku do trwałego przechowywania, np. na dysku twardym.
pythonowy Sposób rozwiązania problemu, który działa elegancko w Pythonie. Np. użycie `try` i `except` jest *pythonowym* sposobem na przywrócenie działania programu w przypadku nieistniejącego pliku.
Quality Assurance Osoba lub zespół skoncentrowany na zapewnieniu ogólnej jakości oprogramowania. QA są często zaangażowani w testowanie i identyfikowanie problemów, zanim produkt zostanie wydany.
znak końca linii Specjalny znak używany w plikach i ciągach znaków do wskazywania końca linii.

7.12. Ćwiczenia

Ćwiczenie 1. Napisz program odczytujący plik i wypisz zawartość pliku (wiersz po wierszu), ale dużymi literami. Uruchomienie programu będzie wyglądało następująco:

```
Podaj nazwę pliku: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

Plik możesz pobrać z adresu https://py4e.pl/code3/mbox-short.txt

Ćwiczenie 2. Napisz program proszący o podanie nazwy pliku, a następnie przeszukaj plik w celu znalezienia linii podobnych do poniższej:

```
X-DSPAM-Confidence: 0.8475
```

Gdy trafisz na linię, która zaczyna się od „X-DSPAM-Confidence:", podziel linię tak, by wyodrębnić z niej liczbę zmiennoprzecinkową. Zliczaj te linie i sumuj wartości oznaczające pewność, że mamy do czynienia ze spamem. Kiedy dotrzesz do końca pliku, wydrukuj średnią wartość pewności spamu.

```
Podaj nazwę pliku: mbox.txt
Średni poziom pewności spamu: 0.894128046745
```

[6] Jeśli korzystamy np. z edytora Atom, to po otwarciu interesującego nas pliku informację o użytym sposobie zapisu oznaczenia końca linii możemy znaleźć na dolnej belce w prawym dolnym rogu aplikacji. LF (ang. *line feed*) oznacza `\n`, a CR (ang. *carriage return*) oznacza `\r`.

```
Podaj nazwę pliku: mbox-short.txt
Średni poziom pewności spamu: 0.750718518519
```

Przetestuj swój program na plikach `mbox.txt` i `mbox-short.txt`.

Ćwiczenie 3. Czasami, gdy programiści się nudzą lub chcą się trochę zabawić, dodają do swojego programu nieszkodliwy tzw. *Easter Egg*[7] (dosł. z ang. *jajko wielkanocne*). Zmodyfikuj swój program, który pyta użytkownika o nazwę pliku tak, by wypisał zabawną wiadomość, gdy użytkownik wpisze w nim nazwę pliku „trele morele". Program powinien zachowywać się normalnie dla wszystkich innych plików, które istnieją lub nie. Oto przykładowe wykonanie programu:

```
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt

Podaj nazwę pliku: missing.txt
Nie można otworzyć pliku: missing.txt

Podaj nazwę pliku: trele morele
TRELE MORELE - co za bzdury!
```

Nie zachęcamy Cię do umieszczania Easter Eggów w Twoich programach; to tylko ćwiczenie.

[7] https://pl.wikipedia.org/wiki/Easter_egg