Język programowania wysokiego poziomu 2 - Wprowadzenie do języka Python

Wprowadzanie danych:
Program zatrzyma się w miejscu, gdzie występuje funkcja input i będzie oczekiwał na wprowadzenie danych.

```
name = input("Enter name:")
```

Przekazywanie parametrów:
Aby przekazać argumenty do skryptu należy zaimportować sys. Następnie można je wykorzystać odwołując się do kolejnych pozycji (odpowiadających kolejnym argumentom) listy argv będącej atrybutem sys. Podobnie jak w przypadku Basha argument zerowy jest nazwą pliku.

```
sys.argv[numer_argumentu]
```

List (albo dict, tuple) comprehensions:
Konstrukcja ta umożliwia tworzenie list przy użyciu znacznie mniejszej ilości kodu. Przykład:

```
my_list = []
for i in range(10):
  my_list.append(i)
```

jest równoważne do:

```
lista = [item for item in range(10)]
```

W takim przypadku w pętli for nie stosujemy dwukropka. Możliwe jest także dodanie warunku, np:

```
lista = [i for iin range(10) if i%2]
```

Analogiczną konstrukcję możemy zastosować do innego typu obiektów, np słowników.
Można to zapamiętać tak:
lista = [wyrażenie for element in kolekcja if warunek]

Obsługa plików tekstowych:
Aby otworzyć plik, używamy wbudowanej funkcji open(). Podajemy ścieżkę do pliku (lub jego nazwę, jeżeli skrypt znajduje się w tym samym katalogu co plik) oraz tryb otwarcia:
“r” - read (odczyt)
“w” - write (zapis, nadpisanie)
“a” - append (zapis, dopisanie)
“x” - create (zwróci błąd, jeżeli plik istnieje)

Dodatkowo można określić, czy plik ma być obsługiwany w trybie binarnym czy tekstowym
„t” – tekst – Wartość domyślna
„b” – binarny (np. obrazy)

Funkcja ta zwraca obiekt, który posiada metodę read() umożliwiającą odczytanie zawartości pliku. Przykład użycia:

```
f = open(“filename.txt”, “r”)
file_content = f.read()
```

Taki sposób korzystania z funkcji open niesie ryzyko pozostawienia niezamkniętego pliku.

Wyjątki:
Wyjątki to sytuacje wyjątkowe, niepożądane (zazwyczaj błędy) napotkane na etapie wykonywania programu. Wystąpienie wyjątku, które określa się także jako wyrzucenie wyjątku (z ang. throw) spowoduje zakończenie działania programu, jeżeli wyjątek nie zostanie obsłużony.

Obsługa wyjątków: wyjątki obsługujemy poprzez zastosowanie bloków try, except i finally. Wiele operacji bibliotecznych wykorzystuje wyjątki, stąd np. odwołanie się do nieistniejącego elementu nie zaowocuje uzyskaniem niezdefiniowanej wartości, a jedynie spowoduje “wyrzucenie” wyjątku, który pozostawiony bez obsługi spowoduje zatrzymanie programu. Pod względem zarządzania pamięcią Python jest językiem bezpiecznym, jednak ma to swoją cenę - wydajność.

```
try:
      # zostanie podjęta próba wykonania kodu w tym miejscu

except:
      # jeżeli kod z bloku try wyrzucił wyjątek, to wykona się ten blok
      # można także zastosować poniższą konstrukcję:
      # except Exception as e:
      # co daje nam dostęp do szczegółów wyjątku, aby potem np. print(e)

finally:
      # ten blok wykona się zawsze, niezależnie od tego, czy wyjątek
      # zostanie wyrzucony
```

Context manager:
Umożliwia przydzielanie i zwalnianie zasobów dokładnie wtedy, gdy tego potrzebujemy. Najpopularniejszy przykład - instrukcja with. Załóżmy, że mamy dwie powiązane operacje, które chcemy wykonać parami, z blokiem kodu pomiędzy nimi. Konstrukcja:

```
with open('file.txt', 'w') as opened_file:
    opened_file.write('Hello!')
```

Jest równoważna do:

```
def make_sequence(first_num, last_num):
  seq = []
  for i in range(first_num, last_num):
    seq.append(i)
  return seq
```

Generatory

To specjalne funkcje, które zachowaniami przypominają iteratory. Najczęściej są używane razem z pętlami for. Rozważmy przypadek, kiedy chcemy wykonać operacje na wartościach od do danej liczby:

```
def make_sequence(first_num, last_num):
  seq = []
  for i in range(first_num, last_num):
      seq.append(i)
  return seq
```

Taka funkcja zwróci pełną listę, w całości wypełnioną pożądanymi wartościami. Co jednak kiedy lista zawiera dużą ilość obiektów, które na dodatek mają ogromny rozmiar? Przekazywanie takiej struktury w całości jest kosztowne obliczeniowo, a czasem nawet trudne technicznie. Z pomocą przychodzą generatory, które pozwalają nam na dostęp jedynie do pożądanych w danym czasie zasobów z danej struktury.

```
def make_sequence(first_num, last_num):
  for i in range(first_num, last_num):
      yield i
```

Należy pamiętać, że wywołana funkcja zwraca generator, dlatego najczęściej chcąc go wykorzystać w kodzie należy wywołać jego wartość (poprzez funkcję next), lub użyć go np z pętlą for:

```
for i in make_sequence(0,20):
  print(i)
```

Można też rzutować generator np. na listę.

Dekoratory

Dekoratory stanowią wsparcie funkcyjnego paradygmatu programowania w języku Python. Funkcja jako obiekt nie tylko może wykonywać operacje, ale sama może być przekazywana jako parametr innych funkcji. Rozważmy następujący przykład:

```
def external(param):
      print("I am just a wrapper. You see this msg before execution of another function")
      value = param()
      print(f"Function has been executed and returned {value}")

def multiply_numbers():
      print("I am internal function and i am drawing the number for you")
      return r.randint(0,9)
```

Widzimy, że jedna funkcja przekazywana jest jako parametr oraz wykonywana wewnątrz drugiej funkcji. Zaprezentowane podejście ma jednak wiele wad. Istnieje jednak dedykowany sposób umożliwiający nam to samo. Są to dekoratory, czyli konstrukcje odpowiadające szczególnym funkcjom. Przykład:

```
def external(param):
    def wrapper():
      print("I am just a wrapper. You see this msg before execution of another function")
      value = param()
      print(f"Function has been executed and returned {value}")
    return wrapper
```

chcąc udekorować funkcję należy poprzedzić jej definicję dekoratorem rozpoczynającym się od symbolu @.

```
@external
def multiply_numbers():
      print("I am internal function and i am drawing the number for you")
      return r.randint(0,9)
```

Biblioteka OS
Jedną z kluczowych bibliotek Pythona do interakcji z systemem operacyjnym jest os. Dokumentacja tej biblioteki dostępna jest tutaj: https://docs.python.org/3/library/os.html
Wiele z funkcji bibliotecznych odpowiada poznanym już poleceniom powłoki. Przykładowo możliwe jest listowanie, tworzenie i usuwanie katalogów, sprawdzanie czy plik istnieje, operacje na ścieżkach.

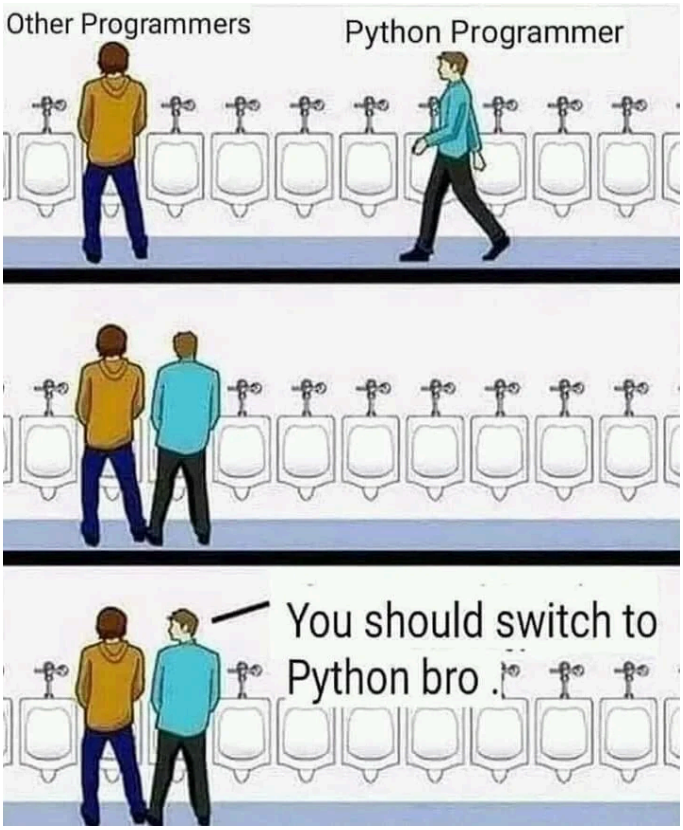
Podstawową zaletą programów napisanych w Pythonie z wykorzystaniem biblioteki os jest ich wieloplatformowość. Polecenia działające na systemach uniksowych/uniksopodobnych nie będą działały na systemie Windows i na odwrót. Podobny problem wystąpi, jeżeli będziemy operować na ścieżkach. Pewną wadą są jednak narzuty związane z przełożeniem kodu Pythona na polecenia powłoki.

Jedną z najczęściej używanych opcji os.path jest możliwość dzielenia i łączenia ścieżek. W tym celu powszechnie stosuje się trzy metody: split, basename i dirname.

W Pythonie znak \ (backslash) zastosowany w łańcuchu tekstowym jest znakiem specjalnym - "escape character", stąd chcąc, aby był widoczny w należy go użyć podwójnie lub skorzystać z raw stringów (np. r“katalog\plik.txt”).

Zadanie1:
Napisz skrypt tworzący n katalogów, a w każdym z nich m katalogów. Wylosuj katalog z poziomu n (poszukaj w Internecie jak się losuje liczby w Pythonie i zaimplementuj prosty mechanizm wyboru katalogu na podstawie wylosowanej liczby) i wstaw do niego plik tekstowy o dowolnej nazwie, którego zawartość będzie Twoim imieniem i nazwiskiem. Następnie napisz kod, który pokaże lokalizację tego pliku.

Zadanie2:
Napisz skrypt, który wczyta plik tekstowy z przykładowymi imionami, nazwiskami i numerem indeksu. Plik taki należy wcześniej utworzyć i wstawić do niego w dane kilku sztucznych studentów. Numer indeksu podziel przez cyfrę podaną przez użytkownika **z wykorzystaniem input**. Zastąp numer indeksu w pliku tak powstałą liczbą.
Uwzględnij sytuację, kiedy użytkownik poda nieprawidłową wartość (np string lub 0) - wtedy pozostaw niezmieniony numer indeksu i poinformuj o tym użytkownika. Koniecznie przetestuj co się stanie jak podasz nieprawidłowy parametr.



źródło reddit