PYTH/ON – Mini Skrót wiadomości z komentarzami ☺

podziękowania dla: Skewoo(nazwa kanału youtube)
(IDE: PyCharm, data wykonania skrótu: 29.08.18r.)

Spis treści:

1. Wypisanie na ekran informacji - funkcja print:

- aby wypisać łańcuch znaków(string) to wystarczy zapisać tekst w cudzysłowach

przykład: print(„To jest przykład wypisania tekstu na ekran”)

! warto zwrócić uwagę, że w PYTHON nie kończymy komend średnikiem jak to jest np. w języku C# czy Java

- aby wypisać liczbę jako faktycznie liczbę a nie „porcję znaków”

przykład: print(50)

2. Zmienne:

Python automatycznie konwertuje zmienne stąd nie musimy definiować typu danych!

Przykład zmiennej przechowującej liczbę:
x = 5

jeżeli chcemy wypisać jej wartość do konsoli korzystamy z funkcji print czyli w ten sposób: print(x)

jeżeli chcielibyśmy sprawdzić jaki typ danych ma zmienna x możemy napisać przykładowo: print(type(x))
(funkcja type zwraca typ danych podanego argumentu)

aby wykonać na tej zmiennej operacje matematyczną moglibyśmy napisać tak:
print(„wynik x*2 = ”, x*2)

ewentualnie moglibyśmy też wynik operacji x*2 zapisać do innej zmiennej i dopiero ją wypisać na ekran, np. tak:

```
y = x * 2
print(y)
```

jeżeli chcielibyśmy zapisać liczbę z przecinkiem to po prostu: deklarujemy zmienną, nadajemy wartość, nie przejmujemy się podaniem typu danych i cieszymy się, że nie musimy dokładać średnika

```
liczba = 5.5      // zadeklarowaliśmy liczbę o typie danych typu float
```

oczywiście do zmiennej możemy również zapisać tekst w taki sposób jak powyżej tylko wartość zmiennej podajemy w cudzysłowach, przykładowo:

```
tekst = „to jest przykładowy tekst”
```

pisząc komendę: print(type(tekst)) pod naszą zmienną tekst otrzymamy informację, że zmienna tekst jest typu STR a STR jest skrótem od STRING czyli łańcucha

mamy możliwość dodania kolejnego ciągu znaków do zmiennej tekst, zrobimy to za pomocą operatora ++ , przykładowo:

tekst = tekst ++ „ i jego dodatek”

3. Funkcja input

Umożliwia wczytywanie wartości od użytkownika. Kompilator napotykając miejsce z funkcją input() czeka aż użytkownik coś wpisze w konsoli.

Przykładowa implementacja funkcji input():

zmienna = input()

po kompilacji skryptu będziemy musieli podać wartość, którą zmienna o nazwie „zmienna” będzie przechowywać.

warto w tym momencie przedstawić tzw. „konwersje danych” – załóżmy, że po linijce w której prosimy o podanie czegoś do „zmienna” mamy fragment:

print(„Wynikiem jest:”, zmienna + 2)

podając dla funkcji input() przykładowo łańcuch: „warunek” wiemy, że otrzymamy błąd. Kompilator nie wie co zrobić z operacją zmienna + 2, ale musimy też pamiętać, że funkcja input pobiera łańcuch(string) zatem podając np. 15 również podajemy łańcuch czyli 15 jest traktowane jak znak 1 i znak 5! To sprawia, że nadal nie możemy dodać do naszej piętnastki liczby 2. Co zatem zrobić? Posłużyć się konwersją danych! Poprawnie zapiszemy:

print(„Wynikiem jest:”, (int)zmienna + 2)

4. Manipulacja łańcuchami:

tekst = „Ala ma kota”

nie musimy wypisywać całego tekstu jeżeli nie chcemy, jak to zrobić? – zobaczmy na przykładach:

aby wypisać na ekran pierwszy znak zmiennej tekst zapiszemy:

```
print(tekst[0])                    // otrzymamy w konsoli A
```

przy okazji, warto pamiętać że w programowaniu numerowanie jest od 0

możemy też przypisać znak bądź znaki do zmiennej, nie musimy ich od razu wypisywać na ekran jeżeli nie chcemy

```
y = tekst[3]                    // zatem pod zmienną y będzie znak s
```

jeżeli np. nie chce nam się liczyć, jaką wartość ma przedostatnik znak to możemy skorzystać z udogodnienia polegającego na podaniu ujemnego „indeksu". Podając ujemną wartość będziemy liczyć pozycję znaków od końca, przykładowo:

```
print(tekst[-1])                // otrzymamy przedostatni znak czyli t
```

aby pobrać bądź wyświetlić „wycinek" tekstu wystarczy podać zakres od którego znaku do którego chcemy otrzymać znaki. Możemy też nie podawać jednego z „widełek". Jeżeli nie podamy początkowej granicy – weźmie od początku. Jeżeli nie podamy końcowej granicy – weźmie do końca. Zobaczmy to na przykładach:

```
print(tekst[0:4])               // podaliśmy obie „widełki" , wyświetli: „la m"

print(tekst[2:])                // podaliśmy lewą stronę,  wyświetli: „a ma kota"

print(tekst[:4])                // podaliśmy prawą stronę, wyświetli: „Ala m"
```

5. Instrukcje warunkowe

Wykorzystujemy je gdy chcemy coś porównać. Przykładowo: jeżeli wynik porównania będzie prawdziwy to robimy blok kodu x a jeżeli nie to robimy blok kodu y.

Przykładowa instrukcja warunkowa:

```
x = input()

if x == 5:
        print(„Podano 5!")
else
        print(„Nie podano 5!")
```

i teraz.. skoro nie ma tu średników, program musi odczytać przecież który blok należy do if a który do else.. jak to robi? Ważne jest aby właśnie te bloki odpowiednio upozycjonować – robiąc wcięcie mówisz kompilatorowi, że dana linijka należy do tego bloku! Przypomniając przy okazji == to operator równości czyli zwraca True jeżeli lewa strona jest równa prawej. Przeciwnym operatorem do niego jest operator oznaczany: !=

Przykład bardziej złożonej instrukcji warunkowej:

```
y = input()

if y > 10:
        print(„podano liczbę > 10”)
elif y < 5:
        print(„podano liczbę < 5 „)
else:
        print(„podano liczbę > 5 ale < 10”)
```

dodaliśmy tutaj blok *elif*. Bloki *elif* są dodatkowymi blokami które są sprawdzane jeżeli poprzednie bloki nie zostały spełnione. W momencie gdy jakiś blok zostanie spełniony, pozostałe są pomijane. W momencie gdy żaden z bloków nie zostanie spełniony to jeżeli zaimplementowaliśmy blok „else”, zostanie wykonany blok „else” – w przeciwnym wypadku nic się nie dzieje. Możesz mieć bloków elif ile chcesz.

Możemy nasze warunki jeszcze bardziej „zawężyć” za pomocą operatorów logicznych. Mamy do dyspozycji: and, or, not

przykład:

```
if x == 5 or x == 2:
        print(„Hello!”)
```

6. Listy

lista nie wymaga podania ile elementów może pomieścić. Możemy na bieżąco dodawać do niej elementy bądź usuwać je.

przykładowa lista:

```
testy = [„mleko”, „ser”, „parowki”]    // listę podajemy w nawiasach kwadratowych
```

aby wypisać element bądź elementy tej listy robimy tak samo jak w przypadku wypisywania „wycinku” bądź jednego znaku tekstu czyli przykładowo:

```
print(testy[0])                 // wypisze mleko(pierwszy element listy)

print(testy[0:1])               // wypisze na ekran: mleko,ser

print(testy[1:])                // wypisze na ekran: ser, parowki
```

aby dodać kolejny element do listy możemy posłużyć się gotową funkcją „append”. Przykladowe jej zastosowanie:

```
testy.append(„wedlina”)
```

jeżeli chcielibyśmy wyczyścić listę:

```
testy.clear()
```

możemy też np. wykorzystać funkcję count do zliczenia danego elementu w liście
print(testy.count(„ser”)) // zwróci 1 bo mamy tylko jeden ser na liście

a jeżeli potrzebujemy rozszerzyć listę o kolejną listę elementów to moglibyśmy to zrobić stosując funkcję „extend”. Przykładowo mamy listę produktyDodatkowe, to żeby rozszerzyć listę „testy” o listę produktyDodatkowe (czyli po prostu dołączenie elementów listy produktyDodatkowe do listy testy) zapiszemy komendę tak:
testy.extend(produktyDodatkowe)

inne przykładowe funkcje dla list:
pop() – usunięcie elementu z listy podając indeks
remove() – usunięcie elementu z listy podając nazwę
index() – zwraca pozycje danego elementu z listy

7. Tuple

różnią się od list tym, że nie można ich edytować tzn. nie możemy do tupli dodać elementów, usuwać. Tuple są wydajniejsze(szybsze) od list ale kosztem braku możliwości edytowania ich.

przykładowa tupla:
warzywa = („ogorek”, „pomidor”) // tuplę podajemy w nawiasach otwartych

8. Słowniki

z definicji: „zestaw kluczy i wartości”. Jeżeli chcemy utworzyć słownik, korzystamy z nawiasów klamrowych!

przykładowy słownik:
Person = {„wiek”: 20, „imie”: „Ania”}
kluczem jest tutaj wiek, imie
wartościami są tutaj 20, Ania
jeżeli chcielibyśmy wypisać element ze słownika zrobilibyśmy to np. tak:
print(person[„imie”])

słownik w odróżnieniu od listy, tupli pozwala nam tworzyć własne klucze czyli zamiast posługiwać się indeksowaniem 0,1,2 możemy posłużyć się kluczami aby odwołać się do żądanej wartości. Dla słownika masz dostępne też gotowe metody jak copy, keys itd..

Podsumowując, do zapamiętania jest, że, stosujemy:
[ ] jeżeli tworzymy listę
( ) jeżeli tworzymy tuplę
{ } jeżeli tworzymy słownik

9. Pętle

petla to fragment kodu który wykonuje sięwięcej razy niż raz

przykładowa pętla while:

```
x = 0

while x <  10:
        print(x)
        x += 1
```

i tu należy też uważać na odstępy – jeżeli jest wcięcie to program interpretuje dana liniję jako fragment bloku while

przykładowa pętla for:

```
lista = [„ab”, „ba”, „aab”]

for slowo in lista:
        print(slowo)
```

przykład pętli for z if + break:

```
lista = [„cc”, „aa”, „bb”]

for i in lista
        if i == „aa”:
                break                    // break powoduje wyjście z pętli
        print(i)
// powyższa pętla for wypisze nam tylko cc
```

przypomnienie: instrukcja continue powoduje pominięcie instrukcji które znajduja się pod nią(tylko dla pętli)

10. Data i czas

Aby móc korzystać z wbudowanych funkcji daty i czasu należy zaimportować moduł o nazwie datetime, robimy to w ten sposób:

```
import datetime
```

dzięki temu możemy korzystać z np.

- ⇨ time.sleep(1) – uśpi program na 1 sekundę
- ⇨ datetime.datetime.now() – moduł ma taką samą nazwę jak obiekt dlatego zapisane jest dwa razy datetime – tu mamy wszelkie informacje o dacie

możemy sobie oczywiście wynik zapisać do zmiennej:
teraz = datetime.datetime.now()

i jeżeli np. chcemy tylko minuty wypisać do konsoli to wystarczy zapisać:
print(str(teraz.minute))

11. Funkcje

część/fragment kodu który spełnia daną czynność lub wykonuje dane zadanie. Funkcje są po to aby zmniejszyć powtarzalność kodu ale też po to aby zwiększyć jego czytelność.

Uwaga, definicję funkcji rozpoczynamy od słówka def(define)

przykładowa funkcja:
```
def printme():
        print(„Hello!!!”)
```
aby wywołąć powyższą funkcję zapiszemy:
printme()

przykład funkcji:
```
def dodaj(a, b):
        return a + b
```

wywołanie: print(dodaj(2,3))

możemy argument funkcji ustawić jako „opcjonalny” czyli nie trzeba będzie go podawać aby prawidłowo wykonać funkcję, robimy to po prostu nadając argumentowi wartość domyślną – przykład:

```
def funkcjaOp(a, b=3):
        return a + b
```

i jeżeli napiszemy: print(funkcjaOp(2)) to kompilator weźmie pod „b” wartość domyślną czyli 3 i bez żadnego błędu otrzymamy wynik 5

12. Własne moduły

W Python możemy stworzyć własny moduł – czyli taki zestaw funkcji dzięki czemu nie musimy ich za każdym razem pisać od nowa/przepisywać tylko importujemy moduł i w łatwy sposób możemy się do niej odwołać.

Aby zaimportować swój własny zestaw funkcji:
import nazwa_modułu

Aby wykorzystać funkcję z modułu który zaimportowaliśmy:
nazwa_modułu.nazwa_funkcji…

przykład:
```
import funkcjeMat

funkcjeMat.dodaj(3,5)
```

A co jeżeli chcemy np. tylko jedną funkcję z modułu zaimportować a nie np. cały moduł gdzie mamy utworzone 500 funkcji?

przykład(chcemy z modułu funkcjeMat zaimportować tylko dodaj i odejmij)
```
from funkcjeMat import dodaj, odejmij
```

I możemy teraz z nich korzystać

13. Obsługa plikow

- aby otworzyć plik, stosujemy polecenie open. Jako pierwsze podajemy nazwę pliku, potem ustawiamy tryb – są dostępne różne tryby otworzenia pliku(w, w+, r, r+, a, a+) np. że możemy do niego zapisywać informacje ale za każdym razem tworzony jest od nowa…

Przykład(chcemy zapisać tekst do pliku plik.txt), utwórzmy w pierwszej kolejności uchwyt do pliku:

```
uchwyt = open(„plik.txt”, „a+”)
```

zapisujemy tekst do pliku:
```
uchwyt.write(„Informacja tekstowa”)
```

i zgodnie z konwencją, zamykamy plik aby można było go edytować, zapisywać przez inne programy czyli:

```
uchwyt.close()
```

14. Moduł OS

czyli Operating System dostarcza nam funkcji pozwalających na operacje na dysku np. tworzenie folderów, listowanie plików itd. Aby móc z niego korzystać:

```
import os
```

⇨ Funkcja do listowania plików i katalogów - listdir
przykład: print(os.listdir(„C:/Windows”)

⇨ Zwraca True jeżeli argument jest plikiem
os.path.isfile(plik)

⇨ Zwraca True jeżeli argument jest katalogiem
os.path.isdir(plik)

⇨ Zmiana nazwy pliku
os.rename(„nazwa_pliku”, „nowa nazwa”)

⇨ Usunięcie pliku
os.remove(„nazwa_pliku”)

⇨ Usuniecie folderu
os.rmdir(„nazwa_folderu”)

⇨ Tworzenie folderu
os.mkdir(„nazwa_folderu”)

15. Wyjątki

Wyjątki warto umieć. W razie problemu z programem chcemy aby nie wyłączał się od razu tylko aby zwrócił informację przez co wystąpił błąd.

Kod który chcemy przetestować umieszczamy w bloku try:

Jeżeli chcemy złapać konkretny wyjątek to piszemy except i nazwa konkretnego wyjątku

Jeżeli chcemy złapać każdy wyjątek to wystarczy except Exception

przykład:

```
try:
        plik = open(„test.txt”, „r”)
        plik.close()
except FileNotFoundError as e:                  // złapie wyjątek „pliku nie znaleziono”
        print(e)
except Exception as e:
        print(„Wystapil nieoczekiwany blad!”)        // złapie każdy wyjątek
```

16. Klasa, obiekt, metody magiczne

KLASA – struktura obiektu(np. człowiek)
OBIEKT – konkretny byt z rzeczywistości(np. Tomek)

Jeżeli tworzymy metodę z „dwoma podłogami” to znaczy że będzie to metoda magiczna – metodę magiczną ma wywoływać tylko PyCharm

Przykład:

```
Class Calculator():      // w nawiasie można zapisać z jakiej innej klasy ma klasa
                         kalkulator dziedziczyć

        // metoda magiczna
        Def __init__(self):              // parametr self mówi „ten konkretny obiekt”
                Self.liczba = 10
```

init – konstruktor
del - destruktor

utworznie obiektu klasy Calculator:
calc = Calculator()