

Wyższa Szkoła Przedsiębiorczości i Administracji



Programowanie
łańcuchy znaków

Prowadzący: dr inż. Sylwester Korga
Własność materiałów edukacyjnych: dr inż. Sylwester Korga

TEMATYKA WYKŁADÓW

Agenda:

1. Wprowadzenie do tematyki programowania.
2. łańcuchy znaków.
3. Instrukcje warunkowe.
4. Pętla for.
5. Pętla while.
6. Listy, tablice, słowniki, krotki, zbiory.
7. Funkcje
8. Pakiety i moduły zewnętrzne.
9. Klasy i obiekty.
10. Tkinter.
11. Jupyter i data science.
12. Obsługa błędów i testowanie.
13. Biblioteka Bpy
14. Bazy danych w programowaniu.
15. Zadania egzaminacyjne

Które języki programowania zaliczane są do języków obiektowych?

Obiektowe języki programowania to te, które umożliwiają programistom definiowanie i używanie obiektów w kodzie. Oto kilka przykładów języków programowania, które są obiektowe:

1. Java,
2. C++,
3. Python,
4. Ruby,
5. C#
6. Objective-C,
7. Swift,
8. Kotlin,
9. JavaScript,
10. Scala,
11. I inne.

Cechy języka Python:

- Interpretowany,
- Interaktywny,
- obiektowo-zorientowany,
- Uproszczony zapis kodu względem innych języków z przejrzystością i łatwością wykonywania złożonych operacji,
- wygodna diagnostyka błędów z ogromnym ekosystemem modułów dla najróżniejszych zastosowań
- łatwość łączenia z kodem w innych językach.
- skalowalność i przenośność.

Cechy języka Python:

- Wspiera wiele paradygmatów programowania,
- Obowiązkowe wcięcia i brak instrukcji skoku,
- Jest pragmatyczny,
- Wiele programów można zapisać bez instrukcji sterujących
- Jest zorientowany obiektowo (OOP),
- Wszystko w Pythonie jest obiektem i można tworzyć własne klasy i obiekty,
- Można natychmiast sprawdzić wynik operacji,
- Jest to język dynamiczny,
- Wspiera wiele paradygmatów programowania.

Cechy języka Python:

- Oprogramowanie z otwartym kodem
- Każdy może brać udział w rozwoju Pythona
- Rozwój koordynuje Python Software Foundation
- Wspierany przez wiele firm członków
- Wspierany w szczególności przez Google
- Autor Pythona Guido van Rossum pracuje dla Google
- Najpopularniejszy język skryptowy w Google
- Stosowany do nauki programowania na wielu uczelniach na całym świecie.

Co to znaczy że język programowania jest dynamiczny?

Język programowania jest dynamiczny, gdy jego typowanie odbywa się w czasie wykonania programu, a nie w czasie kompilacji. Oznacza to, że zmienne w takim języku nie muszą być deklarowane z góry, a ich typ może się zmieniać w trakcie działania programu.

W językach dynamicznych typ zmiennych jest określany na podstawie wartości, jaką przechowuje zmienna, a nie na podstawie deklaracji typu, jak ma to miejsce w językach statycznych. Dynamiczne języki programowania mają także wiele innych cech, takich jak łatwość tworzenia, testowania i modyfikowania kodu oraz zdolność do wykonywania skomplikowanych operacji na danych w czasie rzeczywistym.

Przykład:

```
zmienna1=Input("wpisz wartość")
11
zmienna2=Input("wpisz wartość")
jedenaście
```

W tym przypadku, typ zmiennej jest określany w trakcie przypisywania wartości przez użytkownika. Jeśli zmiennej zostanie przypisana wartość liczbowa, to jej typ zostanie ustawiony na liczbowy (int lub float), a jeśli zostanie przypisany ciąg znaków, to jej typ zostanie ustawiony na typ tekstowy (string).

Jeśli język nie jest dynamiczny to znaczy, że jest statyczny.

Co to znaczy że język programowania jest statyczny?

Stacyjny język programowania to taki język, który wymaga, aby zmienne były zadeklarowane z wyprzedzeniem i określonego typu danych, a także wymaga, aby każda zmienna była zainicjowana przed użyciem.

W statycznym języku programowania, typy danych są sprawdzane w czasie kompilacji, co oznacza, że błędy typów danych zostaną wykryte przed uruchomieniem programu. Dzięki temu można uniknąć wielu błędów programistycznych i uzyskać większą pewność co do poprawności kodu.

Przykłady statycznych języków programowania. to Java, C++, C# oraz Kotlin

Cechy języka Python:

Python jest dostępny na wiele platform sprzętowych:

- Windows
- Linux
- Mac OS X
- Symbian S60 (telefony Nokii)

Jak uważasz, który system operacyjny jest „najlepszy” dla programisty i dlaczego jest to Linux? ☺

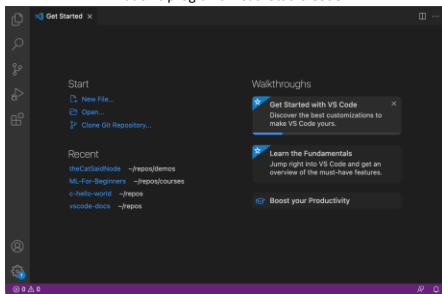
Środowiska programistyczne IDE:

Istnieje wiele różnych środowisk programistycznych IDE (Integrated Development Environment), które oferują narzędzia do pisania, testowania i debugowania kodu.

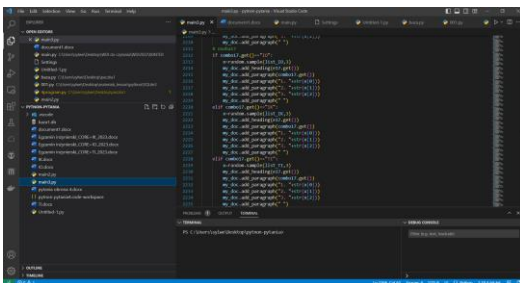
Niektóre z najpopularniejszych środowisk programistycznych IDE to:

1. Eclipse - darmowe i otwarte źródła, zintegrowane środowisko programistyczne dla języków Java, C/C++, PHP i innych.
2. Visual Studio - płatne i bezpłatne wersje, zintegrowane środowisko programistyczne dla języków C++, C#, .NET, TypeScript, Python i innych.
3. NetBeans - darmowe i otwarte źródła, zintegrowane środowisko programistyczne dla języków Java, JavaScript, HTML5, PHP i innych.
4. PyCharm - płatne i bezpłatne wersje, zintegrowane środowisko programistyczne dla języka Python.
5. Sublime Text - płatny edytor kodu źródłowego, który ma wiele przydatnych funkcji i rozszerzeń.
6. Atom - darmowy i otwarte źródła edytor kodu źródłowego z dostępnymi rozszerzeniami i dodatkami.
7. Visual Studio Code - bezpłatny i otwarte źródła edytor kodu źródłowego z funkcjami rozszerzalności i wsparciem dla wielu języków programowania.
8. VIM - ale tam lepiej nie wchodzić ;)

Budowa programu Visual Studio Code:



Budowa programu Visual Studio Code:



Jak sprawdzić jaka wersja języka Python jest zainstalowana?

```
python --version
python3 --version
```

```
PS C:\Users\sylwe\Desktop\pytnon-pytania> python --version
Python 3.10.4
PS C:\Users\sylwe\Desktop\pytnon-pytania>
```

Do czego służy pakiet Anaconda i dlaczego warto go zainstalować?

Anaconda to popularny pakiet do obliczeń naukowych i analizy danych, zawierający zestaw narzędzi i bibliotek umożliwiających pracę z językiem Python. Anaconda zawiera wiele popularnych bibliotek takich jak NumPy, pandas, Matplotlib, SciPy.

Anaconda dostarcza nie tylko narzędzia i biblioteki, ale także menedżer pakietów, który pozwala na łatwe instalowanie i zarządzanie różnymi wersjami bibliotek, co jest szczególnie ważne w projektach wymagających precyzyjnego zarządzania zależnościami.

Anaconda może być zainstalowane na różnych platformach, takich jak Windows, macOS i Linux, i oferuje prosty interfejs graficzny oraz konsolę, która umożliwia łatwe zarządzanie środowiskiem wirtualnym i pakietami.

Inne popularne pakiety to np. pip, Conda, Virtualenv, i Pyenv.

Co to jest zmienna i stała?

Zmienna to nazwany obszar w pamięci komputera, który przechowuje wartość danego typu danych. Zmienna może przyjmować różne wartości w trakcie działania programu i może być modyfikowana w czasie wykonywania programu. Przykładowe typy danych, które mogą być przechowywane w zmiennych, to liczby całkowite, liczby zmiennoprzecinkowe, ciągi znaków, wartości logiczne i wiele innych. Aby utworzyć zmienną, należy zadeklarować jej typ oraz nazwę, np.

```
int liczba = 5; . Nazwa zmiennej to inaczej identyfikator zmiennej.
```

Stała, w przeciwieństwie do zmiennej, jest wartością, która nie ulega zmianie w trakcie wykonywania programu. Stała jest zwykle używana do przechowywania wartości, które są znane przed uruchomieniem programu i są stałe przez cały czas jego działania, np. wartości matematyczne lub wartości, które są ustalane w trakcie konfiguracji programu. Stała jest deklarowana w podobny sposób jak zmienna, ale z dodatkowym słowem kluczowym "const", np.

```
const double PI = 3.14;
```

Czy w języku Python musimy deklarować typ danych?

Jak prawidłowo nazywać zmienne w kodzie programu?

W jaki sposób nadajemy nazwy (identyfikatory) stałym i zmiennym?

W Pythonie zmienne są nazywane zgodnie z konwencją nazewnictwa PEP 8, która sugeruje, że nazwy zmiennych powinny być pisane małymi literami, oddzielone podkreśleniem a jeśli to konieczne, powinny być opisowe.

Przykłady poprawnego nazewnictwa zmiennych w Pythonie:

- 1.zmienna_id zmienna przechowująca liczby całkowite jako id.
- 2.variable_solution zmienna przechowująca rozwiązanie (obliczenie),
- 3.imie0 zmienna przechowująca imię osoby z liczbą na końcu,
- 4.sum_all- zmienna określająca co zawiera,
- 5.lista_zakupow zmienna przechowująca listę z zakupami.

Co to są typy danych w programowaniu?

Typ danych to określony rodzaj informacji, który może być przechowywany w pamięci komputera i przetwarzany przez program. W programowaniu każda zmienna musi mieć określony typ danych, który informuje komputer, jakiej wielkości i jakiego rodzaju informacje ma przechowywać w danym obszarze pamięci.

Przykłady typów danych to:

- liczby całkowite (int),
- liczby zmiennoprzecinkowe (float),
- wartości logiczne (bool),
- ciągi znaków (str),
- listy,
- krotki,
- słowniki,
- itp.

Każdy typ danych ma określone cechy, takie jak zakres wartości, sposób przechowywania w pamięci i operacje, które można na nich wykonywać.

Wybór odpowiedniego typu danych dla danej zmiennej jest ważny, ponieważ ma wpływ na zużycie pamięci i wydajność programu. Przykładowo, gdy chcemy przechowywać liczby całkowite, to używamy typu danych "int", ponieważ zajmuje on mniejszą ilość pamięci niż typ danych "float", który służy do przechowywania liczb zmiennoprzecinkowych.

Jakie typy danych wyróżniamy w programowaniu?

Typ zmiennej określany jest na podstawie typu ostatnio przypisanego wyrażenia

```
a = 1
```

Można go sprawdzić funkcją type

```
type(nazwa zmiennej do sprawdzenia)
```

Zmienne różnych typów zachowują się inaczej w wyrażeniach `+` Operator `+` dodaje liczby, ale łączy napisy

Przykłady praktyczne dotyczące zmiennych i typów:

Do pisania na konsoli służy funkcja print. Przyjmuje ona przez parametr dane do wyświetlenia. Dane te mogą być tekstem, liczbą, datą lub typem złożony. Najprostszy wariant wyglądałby tak:

```
print("wykład prowadzi dr inż. Sylwester Korga")
```

Nie ma znaczenia czy zostaną użyte podwójne apostrofy ", czy pojedyncze ' do zaznaczenia tekstu. Równie dobrze ta instrukcja mogłaby wyglądać tak:

```
print('wykład prowadzi dr inż. Sylwester Korga')
```

Zauważ, że na końcu linii nie ma średnika ani manipulatora strumienia endl;

Uwaga, python inaczej rozpatruje nadanie typu zmiennej pobranej od użytkownika oraz stałej ustalonej w programie.

Przeanalizuj przykład:

```
1 a=input("podaj zmienną a ")
2 print(type(a))
3 b=int(input("podaj zmienną b "))
4 print(type(b))
5 c=input("podaj zmienną c ")
6 print(type(c))
7 d=input("podaj zmienną d ")
8 print(type(d))
9
10 x=43
11 print("typ zmiennej x to typ: ", type(x))
12
```

```
podaj zmienną a 11
<class 'str'>
podaj zmienną b 22
<class 'int'>
podaj zmienną c 33
<class 'str'>
podaj zmienną d 44
<class 'str'>
typ zmiennej x to typ: <class 'int'>
```

Wyższa Szkoła Przedsiębiorczości i Administracji



Wyższa Szkoła
Przedsiębiorczości
i Administracji

Programowanie

Łańcuchy znaków

Prowadzący: dr inż. Sylwester Korga

Własność materiałów edukacyjnych: dr inż. Sylwester Korga

Co to łańcuchy znaków?

W Pythonie, łańcuchy znaków (strings) to sekwencje znaków znajdujące się między cudzysłowami pojedynczymi lub podwójnymi. Łańcuch znaków to układ występujących po sobie znaków. Niekoniecznie musi być to układ liter.

Do zapisu ciągów znaków (łańcuchów) używa się pojedynczego, podwójnego lub potrójnego apostrofa, np.:

```
text = 'Hello, world!'
lub
text = "Hello, world!"
lub
text = """ To jest tekst zapisany
w wielu liniach przy
wykorzystaniu trzech
apostrofów"""
```

Za pomocą potrójnego apostrofu można łączyć łańcuchy znaków i przenosić je do nowej linii.

Czy można wykonywać działania na łańcuchach?

Łańcuchy znaków to sekwencje znaków, takie jak litery, cyfry, znaki specjalne i spacje, które są używane w programowaniu do reprezentowania tekstu. Łańcuchy znaków są jednym z podstawowych typów danych w wielu językach programowania, w tym w językach takich jak Python, Java, C++, JavaScript i wiele innych.

W językach programowania, łańcuchy znaków są zwykle reprezentowane przez ciągi znaków umieszczone między cudzysłowami (np. "Hello, world!"). Łańcuchy znaków mogą być manipulowane i przetwarzane w różny sposób, takie jak łączenie ich, dzielenie, zastępowanie znaków, wyszukiwanie i wiele innych.

Istnieje możliwość wykonywania działań na łańcuchach znaków np. mnożenie przez liczby naturalne np. jeśli zmienna1 = 'ab' to 3*zmienna1 wynosi 'ababab'.

Przypisywanie danych w pythonie
Jak python zapisuje dane?

Tutaj interpreter zaczyna odczyt

$x = 3$

Tutaj interpreter zaczyna odczyt

$x = x + 3$

Ile wynosi x?

Proces deklarowania i definiowania zmiennej tekstowej

Tworzenie zmiennej tekstowej:

```
nazwa_zmiennej1="łańcuch znaków zmiennej"
```

np. aby utworzyć zmienną o nazwie zmienna1, która ma przyjąć łańcuch aabbccdd należy użyć kodu:

```
zmienna1="aabbccdd"
```

Wyświetlamy wartość zmiennej za pomocą wywołania funkcji wbudowanej `print(nazwa_zmiennej1)`.

Różne zmienne mogą mieć te same wartości. Zmienne mogą przechowywać liczby, tekst, listy łańcuchów znaków i tekstów itd.

Słowo zmienna odnosi się w programowaniu do miejsca, w którym przechowywane są informacje.

Funkcje a łańcuchy znaków

W jaki sposób w kodzie poznać, że mam do czynienia z funkcją?

Funkcja posiada nazwę a po prawej stronie nazwy znajdują się nawiasy (). Przykładowe funkcje występujące w Pythonie:

```
print()
len()
count()
max()
append()
```

Jak wydrukować napis w konsoli programu?

```
print('Hello world!')
print("Hello world!")
```

```
print(99)
print(99/3)
```

```
t=[1,2,3,4,5,'czwartek'],[1,2,3,4,5,'czwartek']
```

Bit i Bajt jako jednostki informacji

Bit – jest najmniejszą jednostką w świecie cyfrowym, odpowiada stanowi komórki pamięci. Bit przyjmuje wartość zero lub jeden.

Ciąg ośmiu bitów (czyli zer i / lub jedynek) nazywamy bajtem (byte).

1 BAJT = 8 BITÓW

Bajt to podstawowa jednostka informacji w informatyce, która składa się z 8 bitów. Każdy bit może przyjmować wartość 0 lub 1, co oznacza, że każdy bajt może reprezentować 256 różnych wartości ($2^8 = 256$).

Bit i Bajt jako jednostki informacji

Bajt daje aż 256 różnych kombinacji bitów. Stąd pozwala na zapisanie binarnie liczb od 0 do 255. Innymi słowy każdy bajt to liczba w przedziale od 0 do 255.

W jaki sposób komputer zapisuje litery?

Liczby zapisane binarnie mogą też być interpretowane jako litery.

Przykładowo 0100 0001 odpowiada 65 w systemie dziesiętkowym, co komputer zinterpretuje i wyświetli jako duże A.

```
print(chr(65)) # Wyświetli "A"
```

Funkcja chr() w Pythonie służy do zamiany podanej liczby całkowitej na odpowiadający jej znak Unicode. Zwraca pojedynczy znak o podanej wartości Unicode.

Funkcja ta przyjmuje jeden argument - liczbę całkowitą reprezentującą wartość Unicode znaku. Na przykład: chr(65)

Co to jest kodowanie ASCII?

Czy kodowanie ASCII jest obecnie stosowne?

ASCII to skrót od ang. **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (Amerykański Standardowy Kod wymiany Informacji). Jest to standardowy system kodowania znaków, w którym każdemu znakowi odpowiada unikalna liczba całkowita z zakresu 0-127. W ten sposób, każdy znak tekstu, taki jak litery, cyfry, znaki interpunkcyjne i specjalne, jest reprezentowany przez określoną liczbę w systemie binarnym. ASCII jest powszechnie stosowany w komunikacji między komputerami oraz w innych zastosowaniach, takich jak formatowanie tekstu i kodowanie plików tekstowych.

Jak wygląda tabela ASCII?

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	1		32	20	40		64	40	100	@	96	60	140	
1	1			33	21	41	!	65	41	101	A	97	61	141	a
2	2			34	22	42	"	66	42	102	B	98	62	142	b
3	3			35	23	43	#	67	43	103	C	99	63	143	c
4	4			36	24	44	\$	68	44	104	D	100	64	144	d
5	5			37	25	45	%	69	45	105	E	101	65	145	e
6	6			38	26	46	&	70	46	106	F	102	66	146	f
7	7			39	27	47	'	71	47	107	G	103	67	147	g
8	8			40	28	50	(72	48	110	H	104	68	150	h
9	9			41	29	51)	73	49	111	I	105	69	151	i
10	A			42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B			43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C			44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D			45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E			46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F			47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10			48	30	60	0	80	50	120	P	112	70	160	p
17	11			49	31	61	1	81	51	121	Q	113	71	161	q
18	12			50	32	62	2	82	52	122	R	114	72	162	r
19	13			51	33	63	3	83	53	123	S	115	73	163	s
20	14			52	34	64	4	84	54	124	T	116	74	164	t
21	15			53	35	65	5	85	55	125	U	117	75	165	u
22	16			54	36	66	6	86	56	126	V	118	76	166	v
23	17			55	37	67	7	87	57	127	W	119	77	167	w
24	18			56	38	70	8	88	58	130	X	120	78	170	x
25	19			57	39	71	9	89	59	131	Y	121	79	171	y
26	1A			58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B			59	3B	73	;	91	5B	133	[123	7B	173	[
28	1C			60	3C	74	<	92	5C	134	\	124	7C	174	\
29	1D			61	3D	75	=	93	5D	135]	125	7D	175]
30	1E			62	3E	76	>	94	5E	136	^	126	7E	176	^
31	1F			63	3F	77	?	95	5F	137	_	127	7F	177	_

Co to są strony kodowe?

Strony kodowe to zestawy znaków i ich reprezentacji numerycznych (kodów) używanych do reprezentowania tekstu w danym języku lub grupie języków. W informatyce strony kodowe są używane do mapowania znaków na liczby, które mogą być przechowywane i przesyłane za pomocą systemów komputerowych.

Strony kodowe mają przypisane specjalne numery np.

- Windows-1250 standard środkowoeuropejski,
- Windows-1251 odpowiada cyrylicy,
- Windows-1252 to standard zachodnioeuropejski,
- Windows-1253 koduje greckie znaki,
- Windows-1256 arabskie.

Windows 1250 to strona kodowa opracowana przez firmę Microsoft, która jest używana głównie w systemach Windows do reprezentacji tekstów w językach Europy Środkowej i Wschodniej. Jest to jednobajtowy kod strony, w którym każdy znak jest reprezentowany przez jeden bajt.

W kodowaniu Windows 1250, litera A ma kod szesnastkowy 41 lub dziesiętny 65. Jest to standardowy kod dla litery A w wielu innych popularnych kodowaniach znaków, takich jak ASCII i Unicode.

Strona kodowa Windows 1250 zawiera 256 różnych kodów, z których każdy reprezentuje inny znak lub sekwencję znaków. Obejmuje to litery alfabetu łacińskiego, cyfry, znaki interpunkcyjne, znaki specjalne i wiele innych symboli potrzebnych do reprezentowania tekstów w językach Europy Środkowej i Wschodniej, takich jak polski, czeski, słowacki, węgierski, słoweński, chorwacki i wiele innych.

UNICODE

Unicode to standard kodowania znaków, który pozwala na reprezentację tekstu w różnych językach i systemach pisma na całym świecie.

Unicode obejmuje zestaw znaków z wielu języków, w tym alfabetów, cyfr, znaków interpunkcyjnych, symboli matematycznych, technicznych i innych symboli specjalnych.

Unicode jest rozszerzeniem wcześniejszych standardów kodowania, takich jak ASCII i ISO 8859, które ograniczały się do reprezentacji znaków w określonych językach lub regionach.

Unicode umożliwia jednocześnie reprezentowanie tekstów w różnych językach i systemach pisma na całym świecie, a także ułatwia wymianę informacji między różnymi systemami i aplikacjami.

Kodowanie Unicode może być realizowane za pomocą różnych kodowań zmiennobajtowych i jednobajtowych, takich jak UTF-8, UTF-16, UTF-32, które określają sposób przyporządkowania kodów Unicode do sekwencji bitów.

Czy Unicode i UTF-8 są ze sobą powiązane?

Skrót **UTF-8** oznacza "Unicode Transformation Format - 8-bit". UTF-8 jest jednym z najpopularniejszych kodowań Unicode, które przyporządkowuje każdemu znakowi w standardzie Unicode sekwencję bajtów.

W **UTF-8** każdy znak Unicode reprezentowany jest przez sekwencję od 1 do 4 bajtów. Dzięki temu UTF-8 jest kodowaniem o zmiennym rozmiarze, co oznacza, że różne znaki mogą mieć różne liczby bajtów w zależności od ich kodów Unicode.

UTF-8 jest bardzo popularnym kodowaniem znaków w Internecie i w systemach operacyjnych. Jest to standard kodowania znaków w wielu aplikacjach internetowych, takich jak przeglądarki internetowe, serwery WWW, poczta e-mail, a także w systemach operacyjnych, takich jak Linux, macOS i Windows

Co to jest kodowanie UTF 8?

W programowaniu, kiedy korzystamy z łańcuchów znaków, to właśnie w kodowaniu UTF-8 są one zapisywane w pamięci komputera. Kodowanie to pozwala na poprawne wyświetlanie i przetwarzanie znaków z różnych języków, w tym znaków diakrytycznych, liter alfabetu chińskiego, japońskiego, koreańskiego, czy arabskiego.

Korzystanie z kodowania UTF-8 w programowaniu jest ważne, ponieważ zapewnia ono poprawną obsługę znaków z **różnych języków**, co jest istotne w przypadku tworzenia aplikacji i stron internetowych, które mają być dostępne dla użytkowników z różnych krajów i regionów. Bez poprawnej obsługi kodowania, tekst może być wyświetlany niepoprawnie lub w ogóle nie być widoczny dla użytkowników.

Jak kodowany jest zapis w standardzie UTF-8?

UTF-8 **Hex** to sposób reprezentacji znaków w kodowaniu UTF-8 za pomocą szesnastkowej notacji liczbowej. W UTF-8 **każdy znak jest kodowany w postaci jednej lub więcej sekwencji bajtów**, a każdy bajt może przyjąć wartość od 0 do 255.

W przypadku UTF-8 Hex, każdy bajt jest reprezentowany przez dwie cyfry szesnastkowe, co pozwala na reprezentację każdego bajtu w postaci dwóch znaków. Na przykład, znak "A" w kodowaniu UTF-8 zajmuje tylko jeden bajt i jest reprezentowany przez wartość szesnastkową 41 (dziesiętnie: 65).

Natomiast, bardziej złożone znaki, takie jak chińskie lub japońskie, które wymagają większej liczby bajtów w kodowaniu UTF-8, są reprezentowane przez sekwencje kilku bajtów, z których każdy jest reprezentowany przez dwie cyfry szesnastkowe.

Przykładowo, chiński znak "中" w UTF-8 Hex jest reprezentowany przez sekwencję trzech bajtów: E4 B8 AD (dziesiętnie: 228 184 173). Każdy bajt tej sekwencji jest reprezentowany przez dwie cyfry szesnastkowe.

Wniosek jest taki, że UTF-8 Hex to sposób reprezentacji znaków w kodowaniu UTF-8 za pomocą szesnastkowej notacji liczbowej, co pozwala na łatwe kodowanie i dekodowanie znaków w postaci szesnastkowej.

Jak wygląda tabela kodowania dla UTF 8?

Dec #	Hex	UTF-8 Hex	Char	Unicode description
53	U+0035	35	5	Digit Five
54	U+0036	36	6	Digit Six
55	U+0037	37	7	Digit Seven
56	U+0038	38	8	Digit Eight
57	U+0039	39	9	Digit Nine
58	U+003A	3A	:	Colon
59	U+003B	3B	;	Semicolon
60	U+003C	3C	<	Less-Than Sign
61	U+003D	3D	=	Equals Sign
62	U+003E	3E	>	Greater-Than Sign
63	U+003F	3F	?	Question Mark
64	U+0040	40	@	Commercial At
65	U+0041	41	A	Latin Capital Letter A
66	U+0042	42	B	Latin Capital Letter B
67	U+0043	43	C	Latin Capital Letter C
68	U+0044	44	D	Latin Capital Letter D
69	U+0045	45	E	Latin Capital Letter E
70	U+0046	46	F	Latin Capital Letter F
71	U+0047	47	G	Latin Capital Letter G
72	U+0048	48	H	Latin Capital Letter H
73	U+0049	49	I	Latin Capital Letter I
74	U+004A	4A	J	Latin Capital Letter J
75	U+004B	4B	K	Latin Capital Letter K
76	U+004C	4C	L	Latin Capital Letter L
77	U+004D	4D	M	Latin Capital Letter M
78	U+004E	4E	N	Latin Capital Letter N

Dec #	Hex	UTF-8 Hex	Char	Unicode description
363	U+0168	C3 A8	à	Latin Small Letter U With Macron
364	U+0169	C3 A9	á	Latin Capital Letter U With Breve
365	U+016A	C3 AA	â	Latin Small Letter U With Breve
366	U+016B	C3 AB	ä	Latin Capital Letter U With Ring Above
367	U+016C	C3 AC	ä	Latin Small Letter U With Ring Above
368	U+016D	C3 AD	å	Latin Capital Letter U With Double Acute
369	U+016E	C3 AE	ä	Latin Small Letter U With Double Acute
370	U+016F	C3 AF	ü	Latin Capital Letter U With Umlut
371	U+0170	C3 B0	ü	Latin Small Letter U With Umlut
372	U+0171	C3 B1	ß	Latin Capital Letter W With Circumflex
373	U+0172	C3 B2	ß	Latin Small Letter W With Circumflex
374	U+0173	C3 B3	ÿ	Latin Capital Letter Y With Circumflex
375	U+0174	C3 B4	ÿ	Latin Small Letter Y With Circumflex
376	U+0175	C3 B5	ÿ	Latin Capital Letter Y With Diaeresis
377	U+0176	C3 B6	ÿ	Latin Small Letter Y With Diaeresis
378	U+0177	C3 B7	ÿ	Latin Capital Letter Z With Acute
379	U+0178	C3 B8	ÿ	Latin Small Letter Z With Acute
380	U+0179	C3 B9	ÿ	Latin Capital Letter Z With Circumflex
381	U+017A	C3 BA	ÿ	Latin Small Letter Z With Circumflex
382	U+017B	C3 BB	ÿ	Latin Capital Letter Z With Dot Above
383	U+017C	C3 BC	ÿ	Latin Small Letter Z With Dot Above
384	U+017D	C3 BD	ÿ	Latin Capital Letter Z With Caron
385	U+017E	C3 BE	ÿ	Latin Small Letter Z With Caron
386	U+017F	C3 BF	ÿ	Latin Small Letter Long S
387	U+0180	C3 C0	ß	Latin Small Letter B With Stroke
388	U+0181	C3 C1	ß	Latin Capital Letter B With Stroke
389	U+0182	C3 C2	ß	Latin Capital Letter B With Tilde
390	U+0183	C3 C3	ß	Latin Small Letter B With Tilde
391	U+0184	C3 C4	ß	Latin Capital Letter T One S

Dlaczego w programowaniu obecnie używa się kodowania UTF-8 a nie zaleca się kodowania ASCII?

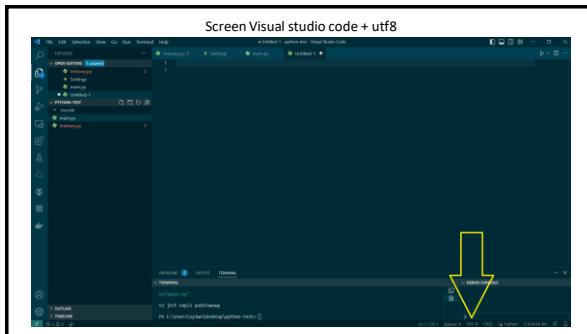
W UTF-8, każdy znak jest reprezentowany przez sekwencję jednego lub więcej bajtów, co pozwala na reprezentację znaków z wielu języków i alfabetów, w tym języków z alfabetem chińskim, japońskim, arabskim, hebrajskim, koreańskim i wiele innych.

W praktyce, tabela UTF-8 obejmuje ponad 1,1 miliona kodów znaków, co daje duże możliwości w reprezentacji wielu znaków z różnych języków i alfabetów. Jednak w rzeczywistości, większość aplikacji i stron internetowych używa tylko niewielkiej części tej tabeli, zawierającej podstawowe znaki z alfabetu łacińskiego, cyfry i znaki specjalne.

Warto również zauważyć, że tabela UTF-8 jest zgodna z tabelą Unicode, co oznacza, że wspiera ona wiele znaków, które nie są reprezentowane w innych tabelach kodowania znaków.

Ile znaków zawiera tabela ASCII?

Tabela ASCII (American Standard Code for Information Interchange) zawiera 128 znaków, które reprezentują podstawowe znaki, cyfry, znaki specjalne i sterujące używane w języku angielskim i innych językach opartych na alfabecie łacińskim.



Podsumowanie informacji o kodowaniu znaków.

W informatyce istnieje wiele standardów kodowania znaków.

ASCII- system kodowania znaków, który stał się prekursorem kolejnych sposobów kodowania.

W kodowaniu UTF-8, pierwsze 128 znaków (od 0x00 do 0x7F) odpowiada kodowaniu ASCII, a więc litera "A" ma taką samą reprezentację w kodowaniu ASCII i UTF-8.

W kodowaniu UTF-8, litera "A" jest reprezentowana przez pojedynczy bajt o wartości dziesiętnej 65 lub wartości szesnastkowej 0x41 lub binarnej.

W systemie binarnym litera A jest reprezentowana przez 8-bitową sekwencję bitów: 01000001.

Formatowanie ciągów znaków

Występują trzy podstawowe sposoby formatowania łańcuchów znaków:

- Łączenie danych za pomocą przecinka

```
print("ten napis nie posiada zmiennych")
print("ten napis posiada zmienna x, która wynosi",x)
```
- Łączenie danych za pomocą funkcji format:

```
print(" Liczba {} oraz liczba {} to liczby naturalne ".format(4,5))
#Cyfra 4 jest przed cyfrą 5
```
- Łączenie danych za pomocą f-stringa:

```
x=3
y=5
print(f"Liczby {x} oraz {y} to liczby naturalne")
#Liczby 3 oraz 5 to liczby naturalne
```

Formatowanie łańcuchów (starego typu czyli procentowe)

```
print("-"*83)
print("|lp.|      Nazwa szczytu      | Wysokość w metrach | Kontynent w którym występuje |")
print("-"*83)
print("|%3d| %23s |%20.2f|%30s|%(1,"Mount Everest",8848,"Azja")|)
print("|%3d| %23s |%20.2f|%30s|%(2,"Aconcagua",6961,"Ameryka Południowa")|)
print("|%3d| %23s |%20.2f|%30s|%(3,"Denali",6195,"Ameryka Południowa")|)
print("|%3d| %23s |%20.2f|%30s|%(4,"Kilimandzaro",5895,"Afryka")|)
print("|%3d| %23s |%20.2f|%30s|%(5,"Elbrus",5642,"Europa")|)
print("-"*83)
```

lp.	Nazwa szczytu	Wysokość w metrach	kontynent w którym występuje
1	Mount Everest	8848.00	Azja
2	Aconcagua	6961.00	Ameryka Południowa
3	Denali	6195.00	Ameryka Południowa
4	Kilimandzaro	5895.00	Afryka
5	Elbrus	5642.00	Europa

ps C:\Users\wyb\Desktop\python-test>

W jaki sposób indeksuje się łańcuchy znaków?

Łańcuch znaków indeksuje się na dwa sposoby:

- Od lewej strony do prawej,
- Od prawej strony do lewej.

Python

0 1 2 3 4 5

-6 -5 -4 -3 -2 -1

Co to jest slicing string?

Slicing string to inaczej proces polegający na wykonaniu wycinka z łańcucha znaków. Wycinek tekstu nazywany jest podłańcuchem lub substringiem.

S[start:stop:step]

Start position End position The increment

Pozycje start i stop są obowiązkowe natomiast pozycja krok jest opcjonalna.

Co to znaczy że element jest mutowalny? Czy łańcuchy znaków są mutowalne?

W programowaniu, **mutowalność (mutability)** odnosi się do możliwości zmiany wartości danego elementu. Element jest mutowalny, jeśli można go zmienić (modyfikować) po utworzeniu. W przeciwieństwie do tego, elementy niemutowalne są stałe i nie można ich zmienić po utworzeniu.

Przykłady elementów mutowalnych to listy (ang. lists) i słowniki (ang. dictionaries) w języku Python, ponieważ można dodawać, usuwać lub modyfikować ich elementy po utworzeniu.

Natomiast ciągi znaków (ang. strings) w języku Python są niemutowalne, ponieważ po utworzeniu nie można zmienić ich zawartości, a jedynie stworzyć nowy ciąg na podstawie istniejącego.

Mutowalność jest ważnym pojęciem w programowaniu, ponieważ wpływa na to, jak programiści projektują i implementują swoje aplikacje. W zależności od potrzeb i wymagań aplikacji, mogą wybierać między mutowalnymi i niemutowalnymi elementami.

Przykłady wykonania podłańcuchów stringa:

```
zmienna1="abcdefghijk"          <start:stop>  <0;4>)

print(zmienna1[0:4])  #abcd
print(zmienna1[1:])   #bcdefghijk
print(zmienna1[:5])   #abcde
print(zmienna1[:])    #abcdefghijk
print(zmienna1[::-1]) #kjihgfedcba
print(zmienna1[-5:-2]) #ghi
print(zmienna1[-10:-4:2])# bdf
print(zmienna1[-4:-10:-2]) #hfd
```

Warto jednak zauważyć, że jeśli krok jest ujemny, to początkowy indeks start musi być większy niż końcowy indeks stop, aby uzyskać wynikowe cięcie o niezerowej długości.

Co to znaczy że element jest mutowalny? Czy łańcuchy znaków są mutowalne?

Przykład niemutowalności łańcucha znaków: (ten kod nie pokaże błędu ale nie zadziała)

```
zmienna1="to jest tekst podstawowy"
zmienna1.upper()
print("zmienna1 po zmianie wygląda tak:",zmienna1)
#zmienna1 po zmianie wygląda tak: to jest tekst podstawowy
```

Ten kod zadziała:

```
zmienna1="to jest tekst podstawowy"
zmienna2=zmienna1.upper()
print("zmienna2 po zmianie wygląda tak:",zmienna2)
#zmienna2 po zmianie wygląda tak: TO JEST TEKST PODSTAWOWY
```

Metody stosowane na łańcuchach znaków

Ponieważ łańcuchy znaków są tak powszechne w programowaniu, wiele języków programowania zapewnia bogaty zestaw funkcji i metod do manipulowania nimi. Te funkcje i metody pozwalają na wykonywanie różnych operacji na łańcuchach znaków, takich jak sprawdzanie ich długości, porównywanie, sortowanie i wiele innych.

Każdy z typów danych posiada swoje metody. Typ string posiada bardzo bogaty zestaw metod do obsługi łańcuchów znaków. Te metody można podzielić na trzy główne grupy:

- Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków.
- Metody zwracające wartość liczbową.
- Metody modyfikowania łańcuchów znaków zwracające wartości logiczne

Metody modyfikowania łańcuchów można podzielić na:

Metody oddające łańcuch znaków

Metody oddające wartość logiczną

Metody oddające liczbę, która jest wynikiem działania tej metody.

Omówmy zatem przykłady metod z każdej przedstawionej grupy

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

lower() Zmienia wszystkie duże litery na małe w stringu

```
zmienna1="TO JEST NAPIS podstawowy"
zmienna2=zmienna1.lower()
print(zmienna1) #TO JEST NAPIS podstawowy
print(zmienna2) #to jest napis podstawowy
```

upper() Zmienia wszystkie małe litery na duże w stringu

```
zmienna1="to jest tekst podstawowy"
zmienna2=zmienna1.upper()
print(zmienna1) #to jest tekst podstawowy
print(zmienna2) #TO JEST TEKST PODSTAWOWY
```

swapcase() Odwraca rodzaj każdej litery – małe na duże, duże na małe

```
zmienna1="TO JEST NAPIS podstawowy"
zmienna2=zmienna1.swapcase()
print(zmienna1) #TO JEST NAPIS podstawowy
print(zmienna2) #to jest napis PODSTAWOWY
```

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

```

capitalize() Zmienia pierwszą literę w ciągu na dużą
zmienna1="to jest napis podstawowy"
zmienna2=zmienna1.capitalize()
print(zmienna1) #to jest napis podstawowy
print(zmienna2) #To jest napis podstawowy

title() Zwraca string – tytuł, w którym wszystkie wyrazy zaczynają się dużą literą, a reszta jest małymi lub są to znaki
nieletterowe
zmienna1="To jest napis podstawowy"
zmienna2=zmienna1.title()
print(zmienna1) #to jest napis podstawowy
print(zmienna2) #To Jest Napis Podstawowy

join(seq) Łączenie (konkatenacja) wyrazów w napisie seq w jeden napis, według separatora/stringu na jakim wywołujemy
metodę. W przykładzie separatorem jest #.
lista1=["1","2","3","4","5"]
zmienna2="#".join(lista1)
print(zmienna2) #1#2#3#4#5
Ciekawostka: Można nie ustawić żadnego separatora i uzyskać ciągłość zapisu:
lista1=["1","2","3","4","5"]
zmienna2="" .join(lista1)
print(zmienna2) 12345

```

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

```

lstrip() Usuwa białe znaki z początku napisu – zwraca kopię pozbawioną białych znaków od lewej strony
zmienna1="  to jest napis podstawowy" #posiada trzy spacje po lewej stronie
zmienna2=zmienna1.lstrip()
print(zmienna1) #  to jest napis podstawowy
print(zmienna2) #to jest napis podstawowy

rstrip() Usuwa białe znaki z końca napisu – zwraca kopię pozbawioną białych znaków od prawej strony
zmienna1="  to jest napis podstawowy " #posiada trzy spacje po prawej stronie
zmienna2=zmienna1.rstrip()
print(len(zmienna1)) #26
print(len(zmienna2)) #23

strip(chars) Usuwa białe znaki lub znak podany jako char z początku i końca napisu – wykonuje lstrip() i rstrip() na napisie.
zmienna1="  to jest napis podstawowy " #posiada trzy spacje po prawej stronie
zmienna2=zmienna1.strip()
print(len(zmienna1)) #30
print(len(zmienna2)) #24

```

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

```

max(string) Zwraca znak o największej wartości w zapisie ASCII
zmienna1="to jest napis podstawowy"
zmienna2=max(zmienna1)
print(zmienna1) #to jest napis podstawowy
print(zmienna2) #y

min(string) Zwraca znak o najmniejszej wartości ASCII
zmienna1="to jest napis podstawowy"
zmienna2=min(zmienna1)
print(zmienna1) #to jest napis podstawowy
print(zmienna2) #  <-----tutaj jest biały znak (spacja). Spacja ma numer 32 a pierwsza
litera alfabetu A posiada numer 65.

split(str="", num=string.count(str)) Dzieli łańcuch według separatora str (spacja jeśli nie podano) i zwraca podciągi jako
listę lub podzieli na co najwyżej liczbę podciągów, jeśli podano num

```

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

`splitlines()` Dzieli cały łańcuch wg znaku nowej linii '\n' i zwraca je jako listę

```
zmienna1="jeden\ndwa \n trzy"
zmienna2=zmienna1.splitlines()
print(zmienna2)
# ['jeden', 'dwa ', ' trzy']

zmienna1 = "Pierwsza linia\nDruga linia\nTrzecia linia"
zmienna2 = zmienna1.splitlines(True)
print(zmienna2) #['Pierwsza linia\n', 'Druga linia\n', 'Trzecia linia']
```

Metody modyfikowania łańcuchów znaków zwracające łańcuchy znaków:

`replace(old, new)` Zamienia wszystkie wystąpienia ciągu `old` na ciąg `new` lub jeśli jest podane `max` – podmiana zostanie wykonana o wskazaną liczbę wystąpień

```
zmienna1="jeden dwa trzy jeden dwa trzy jeden dwa trzy"
zmienna2= zmienna1.replace("jeden", "OSIEM")
print(zmienna2)
#OSIEM dwa trzy OSIEM dwa trzy OSIEM dwa trzy
```

`replace(old, new, max)` Zamienia wszystkie wystąpienia ciągu `old` na ciąg `new` lub jeśli jest podane `max` – podmiana zostanie wykonana o wskazaną liczbę wystąpień

```
zmienna1="jeden dwa trzy jeden dwa trzy jeden dwa trzy"
zmienna2= zmienna1.replace("jeden", "OSIEM",2)
print(zmienna2)
#OSIEM dwa trzy OSIEM dwa trzy jeden dwa trzy
```

Metody zwracające wartości liczbowe

`len(string)` Zwraca długość ciągu znaków

```
zmienna1="to jest napis podstawowy"
zmienna2=len(zmienna1) #24 znaki, spacje też są liczone
```

`count(str, beg=0, end=len(string))` Zlicza ile razy zadany ciąg znaków (`str`) wystąpił w ciągu znaków lub wewnątrz podciągu, który zaczyna się od indeksu `beg` i kończy indeksem `end`

```
zmienna1 = "abc abc abc abc abc"
zmienna2=zmienna1.count("abc",0,7)
print(zmienna2) #2
```

Metody zwracające wartości liczbowe

`find(str, beg=0, end=len(string))` Sprawdza gdzie ciąg str występuje w napisie lub podciągu tego napisu jeśli podamy `index` `beg` i indeks końcowy `end`. Zwraca indeks początkowy lub -1 jeśli ciąg str nie znajduje się w łańcuchu.

```
zmienna1 = "abc abc abc abc"
zmienna2 = zmienna1.find("abc", 0, len(zmienna1))
print(zmienna2) # pokaże index 0
```

`rfind(str, beg=0, end=len(string))` Działa jak `find()`, ale wyszukiwanie od końca ciągu znaków

```
zmienna1 = "abc abc abc abc"
zmienna2 = zmienna1.rfind("abc", 0, len(zmienna1))
print(zmienna2) # Uwaga! Przeszukiwanie stringa odbywa się od prawej strony ale indeks
znalezionej substringa liczony jest od lewej strony.
```

Metody zwracające wartości liczbowe

`index(str, beg=0, end=len(string))` Działa jak `find()`, ale z tą różnicą że zwraca błąd `ValueError` jeśli ciąg str nie zostanie znaleziony

Można wówczas przechwycić błąd za pomocą konstrukcji `try, except`:

```
zmienna1 = "abc abc abc abc"

try:
    zmienna2 = zmienna1.index("def", 0, len(zmienna1))
    print(zmienna2)
except ValueError:
    print("Podłańcuch 'def' nie został znaleziony.")
```

`rindex(str, beg=0, end=len(string))`

Metody modyfikowania łańcuchy znaków zwracające wartości logiczne

Metoda Znaczenie

`isalnum()` Zwraca true jeśli wszystkie znaki w ciągu są alfanumeryczne (litera lub cyfra)

`isalpha()` Zwraca true jeśli wszystkie znaki w ciągu są literami

`isdigit()` Zwraca true jeśli wszystkie znaki w ciągu są cyframi

`islower()` Zwraca true jeśli wszystkie znaki w ciągu są małymi literami.

`isspace()` Zwraca true jeśli wszystkie znaki w ciągu są białymi znakami (spacja, tabulacja, przejście do nowej linii itp)

`istitle()` Zwraca true jeśli ciąg spełnia warunek tytułu (każdy wyraz napisu musi zaczynać się dużą literą i składać wyłącznie z małych liter lub znaków nieliterowych)

`isupper()` Zwraca true jeśli wszystkie znaki w ciągu są dużymi literami.

`startswith(str, beg=0, end=len(string))` Zwraca wynik sprawdzenia, czy napis jest rozpoczęty ciągiem str. Przy podaniu indeksu

`beg`, sprawdzenie rozpoczyna się od tego znaku. Przy wystąpieniu argumentu `end` sprawdzenie zakończy się na tym znaku

`endswith(str, beg=0, end=len(string))` Zwraca wynik sprawdzenia, czy napis jest zakończony ciągiem str. Przy podaniu

indeksu `beg`, sprawdzenie rozpoczyna się od tego znaku. Przy wystąpieniu argumentu `end` sprawdzenie zakończy się na tym

znaku

Zadanie:

Sprawdź czy podane zdanie lub wyrażenie jest palindromem

```
word1=input("write a palindrom to check:")
word2=word1[::-1]
if word1==word2:
    print("it is palindrom")
else:
    print("not")
```

Zmiana typów zmiennych czyli rzutowanie zmiennych czyli.... konwersja

Zmiennej można przypisać wartość pustą None.

None nie jest tożsama z wartością 0, ponieważ oznacza brak wartości, a nie liczbę o wartości 0. Stosuje się ją gdy chcemy zresetować jakąś zmienną lub zdefiniować zmienną bez ustalania jej wartości (wiemy, że zmienna będzie potrzebna później w programie, ale chcemy już na początku zdefiniować wszystkie zmienne).

Jeśli x jest łańcuchem zawierającym liczbę całkowitą np. x='100'.

Aby przekształcić łańcuch na liczbę całkowitą używamy funkcji int np int(x).

x w tej chwili jest typu string

x=int(x)

x w tej chwili jest typu całkowitego

Wyższa Szkoła Przedsiębiorczości i Administracji



Programowanie

Instrukcja warunkowa IF oraz instrukcja #switch(case)

Prowadzący: dr inż. Sylwester Korga

Własność materiałów edukacyjnych: dr inż. Sylwester Korga

Co to są instrukcje w programowaniu? Jakie rodzaje instrukcji się wykorzystuje w kodzie?

Instrukcje (ang. *statement*) możemy generalnie podzielić na instrukcje:

- deklaracyjne
- instrukcję pustą
- grupujące
- wyrażeniowe
- warunkowe (if, if...else)
- wyboru (switch)
- iteracyjne (for, while, do)
- zaniechania (break)
- kontynuowania (continue)
- skoku (goto)
- powrotu (return)
- obsługi wyjątków (try, throw, catch)

Instrukcje warunkowe if

Instrukcja warunkowa (ang. *conditional statement*) to struktura kodu stosowana do podejmowania decyzji.

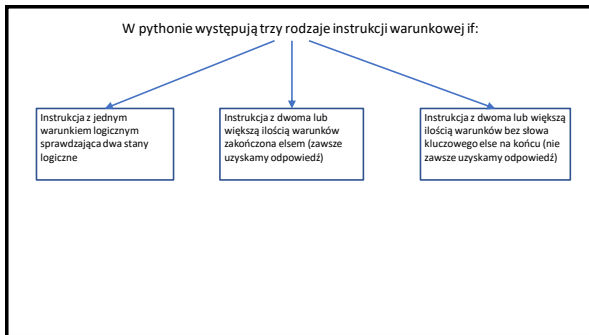
Instrukcja warunkowa to instrukcja w programie, która określa, czy określony blok kodu zostanie wykonany, czy nie. Pozwala programowi podejmować decyzje w oparciu o określone warunki i odpowiednio wykonywać różne bloki kodu. Instrukcje warunkowe są podstawowym elementem składowym programowania komputerowego i są używane w wielu różnych kontekstach, w tym w tworzeniu stron internetowych, analizie danych i obliczeniach naukowych.

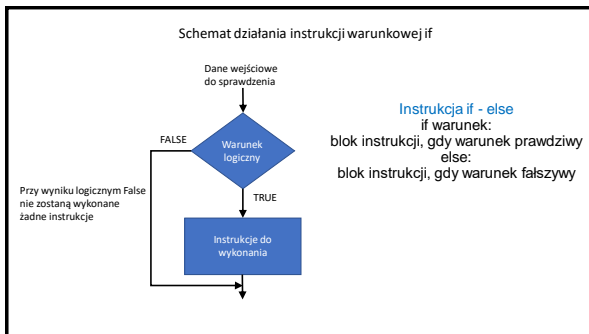
Istnieje kilka różnych sposobów wyrażania instrukcji warunkowych w językach programowania. Najczęstszym sposobem jest użycie instrukcji „if”, która ma zróżnicowaną składnię w zależności od rozpatrywanego problemu.

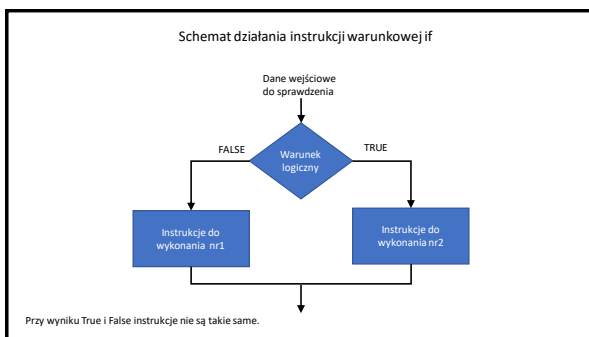
Co to znaczy że program zawiera logikę?

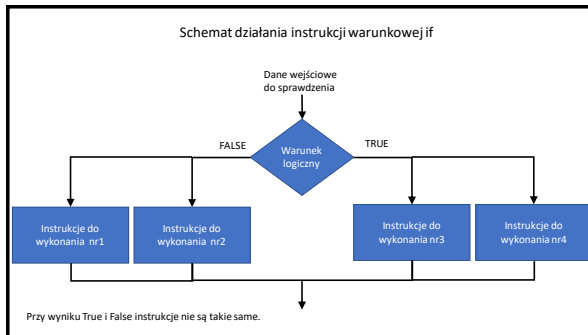
To znaczy, że program zawiera jeden z następujących modułów:

- Wyrażenia logiczne
- Obliczenia wykonywane w zależności od warunków
- Programowanie funkcjonalne
- Instrukcje warunkowe
- Polecenia wykonywane w zależności od warunków
- Programowanie strukturalne
- Obsługa wyjątków
- Reakcja na komunikat o błędzie
- Programowanie obiektowe









Jak wygląda przepływ informacji w strukturze if else:

Instrukcja if i elif

```

if warunek1:
    blok instrukcji, gdy warunek1 prawdziwy
elif warunek2:
    blok instrukcji, gdy warunek2 prawdziwy
elif warunek3:
    blok instrukcji, gdy warunek3 prawdziwy
#tych instrukcji może być nieskończenie wiele (teoretycznie)
else:
    blok instrukcji, gdy żaden z powyższych warunków nie jest
    spełniony
  
```

Nie piszemy instrukcji zagnieźdzonych „if w ifie” pomimo tego, że taka instrukcja będzie działać.

Zamiast zastosować not w:

```

if not warunek:
    instrukcja nr 1
else:
    instrukcja nr 2
  
```

Można zamienić kolejność instrukcji:

```

if warunek:
    instrukcja nr 2
else:
    instrukcja nr 1
  
```

W programowaniu staramy się nie zagnieżdżać instrukcji warunkowych

Nieprawidłowe rozwiązanie

```
if warunek_1:
    if warunek_2:
        instrukcja nr 1
    else:
        instrukcja nr 2
else:
    instrukcja nr 2
```

Prawidłowe rozwiązanie

```
if warunek_1 and warunek_2:
    instrukcja nr 1
else:
    instrukcja nr 2

# warunek 2 będzie
sprawdzany tylko wtedy gdy
warunek 1 jest prawdą
```

Kiedy w pythonie na końcu linii znajduje się dwukropek?

Instrukcja if składa się ze słowa kluczowego if, po którym podawany jest warunek oraz dwukropek:

Wiersze po dwukropku muszą znajdować się w bloku oddalonym od lewej strony edytora o 4 spacje lub 1 tabulator.

```
if warunek_logiczny:
    instrukcja nr 1
    instrukcja nr 2
    instrukcja nr 3
else:
    instrukcja nr 4
    instrukcja nr 5
    instrukcja nr 6
```

4spacje →

1tab →

Instrukcja warunkowa if z jednym warunkiem

Tego typu instrukcja rozpatruje warunki logiczne w których mamy dwa przypadki: tak oraz nie (True oraz False)

Warunek może być zapisany przy instrukcji warunkowej:

```
if warunek_logiczny:
    instrukcja nr 1
    instrukcja nr 2
    instrukcja nr 3
else:
    instrukcja nr 4
    instrukcja nr 5
    instrukcja nr 6
```

Warunek może być przyjęty od zmiennej:

```
warunek=True
if warunek:
    instrukcja nr 1
    instrukcja nr 2
    instrukcja nr 3
else:
    instrukcja nr 4
    instrukcja nr 5
    instrukcja nr 6
```

Przykład wykorzystania instrukcji warunkowej dla jednego warunku:

Zadanie 1:

Sprawdź czy liczba podana przez użytkownika jest dodatnia czy ujemna.

```
number1_user= int(input("podaj liczbę dodatnią lub ujemną"))
if number1_user>0:
    print("podana liczba jest dodatnia")
else:
    print("podana liczba jest ujemna")
```

Pytanie dodatkowe:

Co się stanie jeśli użytkownik poda liczbę zero? Zero nie jest ani dodatnie ani ujemne?

Wówczas musimy rozpatrzyć trzy przypadki.

Operatory logiczne

W Pythonie operatory logiczne są używane w instrukcjach warunkowych, a ich wyniki mogą być przechowywane w zmiennych boolowskich. Jest tak fajne, że używane są słowa kluczowe zamiast kombinacji pojedynczych znaków.

Zadanie 2:

Sprawdź czy liczba podana przez użytkownika jest dodatnia czy ujemna czy równa zero.

```
number1_user= int(input("podaj liczbę dodatnią lub ujemną"))
if number1_user>0:
    print("podana liczba jest dodatnia")
elif number1_user<0:
    print("podana liczba jest ujemna")
else:
    print("podana liczba jest równa zero")
```

#po instrukcji elif koniecznie musi wystąpić warunek

#po instrukcji else nie zamieszczamy warunku

Najczęściej stosowane operatory logiczne:

Koniunkcja (AND) Koniunkcja to zdanie złożone, które jest prawdziwe wtedy i tylko wtedy, gdy wszystkie zdania (warunki) są prawdziwe. Niezależnie od tego, czy jest to warunek jeden, dwa czy osiem, WSZYSTKIE MUSZA być spełnione BEZ WYJĄTKU, aby koniunkcja była prawdziwa. W Pythonie rozpoznajemy koniunkcję po słowie kluczowym „and” (oraz), np.

```
print(True and False) # zwraca False
```

Alternatywa (OR) Alternatywą jest zdanie logiczne oparte na fakcie, że wystarczy jeden spełniony warunek (zdanie), aby były prawdziwe. Słowo kluczowe „or” (lub) to wszystko, czego potrzebujesz, aby zidentyfikować wybór w kodzie, np.

```
print(True or False) # zwraca True
```

Negacja (NOT) Jednoargumentowy operator logiczny umożliwia zastąpienie wyniku twierdzenia jego przeciwieństwem (negacją). W praktyce oznacza to, że każde zdanie złożone jest „odwrotne”, a wtedy każda „prawda” jest „fałszem”, a każdy „fałsz” jest „prawdzy”. W języku angielskim jest to słowo „not” (nie) i po tym samym słowie można bez problemu znaleźć w kodzie negację.

```
print(not True) # zwraca False
```

Nie omawiam operatora XOR, tj. alternatywy wykluczającej. W przeciwieństwie do innych języków, Python tego nie ma. Może dlatego, że jest rzadko używany, ale to tylko moje przypuszczenia.

W Pythonie odstęp od lewego marginesu zastępuje układ nawiasów znany z innych języków programowania. Kod znajdujący się w tej samej odległości od lewego marginesu tworzy blok i za każdym razem, wiersz rozpoczynany jest większą liczbą spacji niż poprzedni, tworzony jest nowy blok, będący częścią poprzedniego bloku.

Wyróżnia się następujące operatory komparacyjne w instrukcjach:

```
== równa się
!= różny od
>= większy lub równy
<= mniejszy lub równy
> większy
< mniejszy
```

Co to są operatory logiczne?

Operatory logiczne działają nie tylko na wartościach logicznych ☐

Napis pusty i liczba zero traktowane są jako fałsz ☐ Pozostałe wartości jako prawda

☐ Operatory logiczne wyliczają wartość drugiego wyrażenia tylko gdy jest to konieczne

- True or p, False and p
- Wartość p nie jest wyliczana
- False or p, True and p
- Wartość p jest wyliczana

Przemienność tylko gdy jeden argument ma wartość fałsz

" and 'a' = "",
 'a' and "" = "",
 " or 'a' = 'a',
 'a' or "" = 'a'

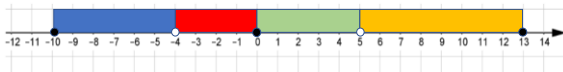
W przeciwnym przypadku brak przemienności

'a' and 'b' = 'b',
 'b' and 'a' = 'a',
 'a' or 'b' = 'a',
 'b' or 'a' = 'b'

W jaki sposób zastosować instrukcję warunkową jeśli rozpatrywanych przypadków jest więcej niż trzy.

Zadanie 3:

Sprawdź w którym przedziale znajduje się liczba podana przez użytkownika. Użytkownik podaje liczbę od -10 do 13.



Przedziały:

A: <-10;4)

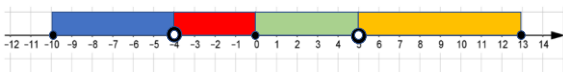
B: <-4;0>

C: (0;5)

D: <5;13>

Zadanie 3:

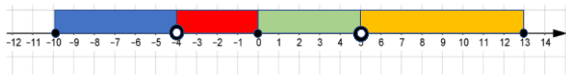
Sprawdź w którym przedziale znajduje się liczba podana przez użytkownika.



Przedziały:

```
number1_user= int(input("podaj liczbę od -10 do 13"))
if number1_user>=-10 and number1_user<4:
    print("podana liczba jest w przedziale A")
elif number1_user>=-4 and number1_user<=0:
    print("podana liczba jest w przedziale B")
elif number1_user>0 and number1_user<5:
    print("podana liczba jest w przedziale C")
else:
    print("podana liczba jest w przedziale D")
```

Jak będzie wyglądała struktura kodu jeśli użytkownik może podać dowolną liczbę całkowitą? Wówczas trzeba rozbudować strukturę kodu o kolejnego elif



```
Przedziały:
number1_user= int(input("podaj dowolną liczbę całkowitą"))
if number1_user>=-10 and number1_user<-4:
    print("podana liczba jest w przedziale A")
elif number1_user>=-4 and number1_user<0:
    print("podana liczba jest w przedziale B")
elif number1_user>0 and number1_user<5:
    print("podana liczba jest w przedziale C")
elif number1_user>5 and number1_user<13:
    print("podana liczba jest w przedziale D")
else:
    print("podana liczba jest nie należy do żadnego przedziału")
```

Jaka będzie funkcjonalność kodu jeśli else nie będzie występował?

```
number1_user= int(input("podaj dowolną liczbę całkowitą"))
if number1_user>=-10 and number1_user<-4:
    print("podana liczba jest w przedziale A")
elif number1_user>=-4 and number1_user<0:
    print("podana liczba jest w przedziale B")
elif number1_user>0 and number1_user<5:
    print("podana liczba jest w przedziale C")
elif number1_user>5 and number1_user<13:
    print("podana liczba jest w przedziale D")
else:
    print("podana liczba jest nie należy do żadnego przedziału")
```

UWAGA: wszystkie warunki po kolei mogą zostać sprawdzone ale żadna instrukcja się nie wykona. Dzieje się tak w sytuacji w której żaden warunek nie będzie spełniony.

Importowanie bibliotek z instrukcją warunkową
Jak to zrobić NIEPRAWIDŁOWO???

Kod pliku biblioteka:

```
def dodawanie(a,b):
    print(a+b)
```

```
dodawanie(3,4)
```

Uruchamiamy program główny gdzie jest tylko print. Co się wyświetli?

Tutaj jest tylko import biblioteki1

7

Dlaczego ta liczba się tutaj pojawiła skoro ona pochodzi z biblioteki? Czy to oznacza, że funkcje z biblioteki same się uruchamiają w bibliotece zewnętrznej? Tak!

Kod programu:

```
import biblioteka
print("Tutaj jest tylko import biblioteki1")
```

Blokowanie funkcji za pomocą instrukcji warunkowej

W Pythonie zmienna `__name__` jest specjalną zmienną, która zawiera nazwę bieżącego modułu. Jej zachowanie zależy od tego, czy plik jest uruchamiany jako program główny, czy jest importowany jako moduł w innym programie.

Gdy plik jest uruchamiany jako program główny, `__name__` przyjmuje wartość `'__main__'`.

Gdy plik jest importowany jako moduł w innym programie, `__name__` przyjmuje nazwę modułu (czyli nazwę pliku bez rozszerzenia .py).

To umożliwia programowi wykrycie, czy jest uruchamiany jako główny program, czy jest importowany jako moduł. Na podstawie tej zmiennej można wprowadzić różne logiki w zależności od kontekstu uruchomienia.

Jak to zapisać? `if __name__ == "__main__":`

Gdzie to zapisać? W pliku funkcji

Blokowanie funkcji z innej biblioteki

Zmienna specjalna `__name__` posiada dwie różne wartości. Pierwsza wartość to `__main__`, druga wartość to nazwa biblioteki z której ta zmienna została zaimportowana.

Kod pliku biblioteka1:

```
print(__name__)
```



Uruchamiamy plik biblioteki.
Co się wyświetli?

`__main__`

Kod programu:

```
import biblioteka
print(__name__)
```



Uruchamiamy plik programu.
Co się wyświetli?

`biblioteka1`
`__main__`

Jeżeli zmienna `__name__` pochodzi z innego pliku to jej wartość zmienia się na nazwę biblioteki z której pochodzi. (Zwróć uwagę na kolory)

Importowanie bibliotek z instrukcją warunkową

Jak to zrobić PRAWIDŁOWO???

Kod biblioteki:

```
def dodawanie(a,b):
    print(a+b)

if __name__ == "__main__":
    dodawanie(3,4)
```

Kod programu:

```
import biblioteka
biblioteka.dodawanie(4,5)
```

Instrukcja switch case czy match case?

W języku Python w wersji 3.10 wprowadzono nową instrukcję "match", która umożliwia dopasowywanie wzorców do wartości. Instrukcja "match" działa podobnie jak instrukcja "switch" znana z innych języków programowania.

Oto ogólna składnia instrukcji "match" w Pythonie:

```
match <wyrażenie>:
    case <wzorzec1>:
        # kod, który zostanie wykonany, gdy <wyrażenie> pasuje do <wzorzec1>
    case <wzorzec2>:
        # kod, który zostanie wykonany, gdy <wyrażenie> pasuje do <wzorzec2>
    ...
    case <wzorzecN>:
        # kod, który zostanie wykonany, gdy <wyrażenie> pasuje do <wzorzecN>
    case <_>:
        # kod, który zostanie wykonany, gdy <wyrażenie> nie pasuje do żadnego wzorca w tym
        # przypadku zastosowano wildcard.
```

Co to jest wildcard?

Wildcard to znak lub sekwencja znaków, która reprezentuje inny znak lub sekwencję znaków w zapytaniu wyszukiwania lub języku programowania.

Wildcardy są często stosowane w zapytaniach wyszukiwania lub wyrażeniach regularnych, aby dopasować szeroki zakres możliwych wartości.

Na przykład, w zapytaniu wyszukiwania plików na komputerze, gwiazdka (*) może być użyta jako wildcard, aby reprezentować dowolną sekwencję znaków. Tak więc, zapytanie wyszukiwania plików o rozszerzeniu „.txt” można zapisać jako „*.txt” przy użyciu gwiazdki jako wildcard.

Przykładowe zastosowanie struktury match

```
number1= int(input("podaj numer"))
match number1:
    case 1:
        print("jeden")
    case 2:
        print("dwa")
    case 3:
        print("trzy")
    case 4:
        print("pięć")
```

Kiedy match trafi na odpowiedni przypadek nie sprawdza już następnych. Zauważ, że koniec każdego z „kejsów” nie jest zakończony instrukcją break tak jak to miało miejsce w innych językach programowania.

Instrukcja match case a warunki logiczne

Instrukcja match tak samo jak instrukcja warunkowa może weryfikować warunki na zdaniach logicznych:

```
number1= int(input("podaj numer"))
match number1:
    case 1:
        print("jeden")
    case 2:
        print("dwa")
    case 3:
        print("trzy")
    case 4:
        print("pięć")
```

Kiedy match trafi na odpowiedni przypadek nie sprawdza już następnych.

Co zastosować w kodzie if czy match case?

Można powiedzieć, że konstrukcja match case potrafi realizować takie same założenia jak instrukcja if ale jest prostszy w zapisie i bardziej czytelny.

```
number1= int(input("podaj numer"))
match number1:
    case 1:
        print("jeden")
    case 2:
        print("dwa")
    case 3:
        print("trzy")
    case 4:
        print("pięć")

number1= int(input("podaj numer"))
if number1==1:
    print("jeden")
elif number1==2:
    print("dwa")
elif number1==3:
    print("trzy")
elif number1==4:
    print("cztery")
elif number1==5:
    print("pięć")
```

A może lepiej zastosować hybrydę if-match case?

A to tak można?

Jak wygląda hybryda if-match case?

Przykładowy program wykorzystujący hybrydę if-matchcase:

```
number1= int(input("podaj numer1"))
number2= int(input("podaj numer2"))
match number2:
    case 1 if number2<50:
        print("jeden")
    case 2 if number2==50:
        print("dwa")
    case 3 if number2>50:
        print("trzy")
    case 4:
        print("pięć")
```

Brak znaku logicznego oznacza operację logiczną AND. Czy można w match case używać operatorów logicznych?

Czy można stosować operatory logiczne w match case?

```
number1= int(input("podaj numer1"))
number2= int(input("podaj numer2"))
match number1:
    case 1 and number2 < 400:
        print("jeden")
    case 2 and number2 == 50:
        print("dwa")
    case 3 or number2 > 50:
        print("trzy")
    case 4:
        print("pięć")
```

Niestety taki zapis jest nieprawidłowy. Wyświetlony zostanie błąd braku znaku :

Jakie dodatkowe możliwości daje instrukcja wyboru match case?

Za pomocą match-case można porównywać zbiory

```
zbiór1=[1,2,3,4,5]
match zbiór1:
    case [1,1,1,1,1]:
        print("to jest zbiór pierwszy")
    case [2,2,2,2,2]:
        print("to jest zbiór drugi")
    case [1,2,3,4,5]:
        print("To jest zbiór trzeci ")
```

Jaki wynik zostanie wyświetlony?

„To jest zbiór trzeci”

Ciekawostka: Instrukcja match case nie rozróżnia rodzajów zbiorów. Nie odróżnia listy od tupli ale wykorzystuje wildcard.

Pamiętasz co oznaczało wildcard?

Jakie dodatkowe możliwości daje instrukcja match-case?

Match case obsługuje wildcard. A to oznacza, że można pracować z wyrażeniami regularnymi ☺

```
zbiór1=[1,2,3,4,5]
match zbiór1:
    case [1,1,1,1,1]:
        print("to jest zbiór pierwszy")
    case [2,2,2,2,2]:
        print("to jest zbiór drugi")
    case [1,2,_,_,_]:
        print("To jest zbiór trzeci ")
```

Jak taki zapis będzie rozpatrywany przez interpreter?

Pierwszy znak to jedynka, drugi znak to dwójka, trzeci, czwarty i piąty znak może być dowolny. W tym przypadku również zostanie wyświetlony napis „To jest zbiór trzeci”

Jak zostanie zinterpretowany przypadek z samymi „wildcards”?

```
zbior1=[1,2,3,4,5]
match zbior1:
    case [1,1,1,1,1]:
        print("to jest zbiór pierwszy")
    case [2,2,2,2,2]:
        print("to jest zbiór drugi")
    case [1,2,_,_,_]:
        print("To jest zbiór trzeci ")
    case [_,_,_,_,_]:
        print("To jest zbiór czwarty")
```

Tutaj również zostanie wyświetlony napis „To jest zbiór trzeci”

Match case potrafi rozpakować zbiory.

Co to znaczy rozpakować zbiór?

```
zbior1=[1,2,3,4,5]
match zbior1:
    case [a,b,c,d,e]:
        print(f"rozpakowane zmienne to po kolei {a},{b},{c},{d},{e}")
```

Napis który zostanie wyświetlony:
rozpakowane zmienne to po kolei 1,2,3,4,5

Match case potrafi również pakować zbiory

Co to znaczy spakować zbiór?

```
zbior1=[1,2,3,4,5,6,7,8,9,10]
match zbior1:
    case [_,_,_,_,*paczka]:
        print(f"w paczce znajdują się następujące elementy {paczka}")
```

Co znajduje się w spakowanej paczce?
W paczce znajdują się następujące elementy [4,5,6,7,8,9,10]
Zwróć uwagę, że ta paczka to lista.

Wyższa Szkoła Przedsiębiorczości i Administracji



Programowanie

Pętla for

Prowadzący: dr inż. Sylwester Korga
Własność materiałów edukacyjnych: dr inż. Sylwester Korga

Pojęcie pętli w programowaniu

Spojrzenie ogólne na pętle:

Pętle w programowaniu służą do wykonywania pewnej sekwencji instrukcji w sposób wielokrotny. Można używać pętli, aby powtórzyć jakiś fragment kodu określoną liczbę razy lub w celu przeiterowania po elementach sekwencji (np. listy, stringa, słownika).

Istnieją dwa główne rodzaje pętli:

- `for`
- `while`.

Pętle są bardzo przydatnym narzędziem w programowaniu, pozwalającym na automatyzację powtarzających się zadań oraz na łatwe przetwarzanie danych zapisanych w sekwencjach.

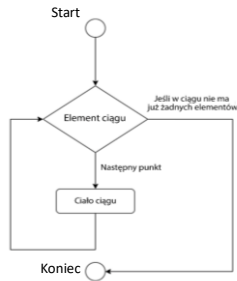
Pętla for – iterowanie po obiektach

Co to są obiekty iterowalne (iterable):

W Pythonie, obiekty sekwencyjne to obiekty, które przechowują sekwencje elementów w określonym porządku. Do najczęściej używanych obiektów sekwencyjnych w Pythonie należą:

1. Listy (lists): listy to obiekty sekwencyjne, które przechowują kolekcję wartości w określonym porządku. Listy mogą przechowywać wartości różnego typu i są mutowalne, co oznacza, że ich wartości mogą być zmieniane po utworzeniu.
2. Krotki (tuples): krotki to obiekty sekwencyjne, które są bardzo podobne do list, z tą różnicą, że są niezmiennie (immutable), co oznacza, że po utworzeniu ich wartości nie mogą być zmieniane.
3. Ciągi znaków (strings): ciągi znaków to obiekty sekwencyjne, które przechowują sekwencje znaków w określonym porządku. Ciągi znaków są niezmiennie (immutable).
4. Bufory (bytearrays): bufory to mutowalne obiekty sekwencyjne, które przechowują sekwencje bajtów w określonym porządku.
5. Zakresy (ranges): zakresy to obiekty sekwencyjne, które przechowują sekwencje liczb całkowitych w określonym porządku.
6. Iteratory (iterators): pętla `for` może być użyta z dowolnym obiektem, który implementuje protokół iteracji, czyli ma metodę `__next__()` zwracającą kolejny element w sekwencji.

Pętla for – schemat przepływu danych



Schemat iterowania
po obiektach za pomocą funkcji for

Zalety stosowania pętli for

Do zalet stosowania pętli for można zaliczyć:

- Iterowanie po elementach sekwencji (np. listy, stringa, słownika).
- Filtrowanie danych.
- Wykonywanie pewnych operacji na każdym elemencie sekwencji.
- Sprawdzanie wszystkich możliwych kombinacji dwóch lub więcej zmiennych.
- Pobieranie od użytkownika danych wielokrotnie, aż do momentu, gdy użytkownik poda poprawne dane.
- Generowanie prostych animacji lub interaktywnych programów.
- Brak powtarzalności kodu
- Automatyzacja kodu

Pętla for - listy

Poniższe kody są takie same. Domyślny argument end przyjmuje wartość \n co oznacza przejście do nowej linii.

```

my_list = [1, 2, 3, 4, 5]    my_list = [1, 2, 3, 4, 5]
for element in my_list:    for element in my_list:
    print(element)          print(element, end="\n")
  
```

Co zostanie wyświetlone?

1
2
3
4
5

W jaki sposób wyświetlić liczby w jednej linii?

Można zmienić wartość parametru end na end=" " #to jest znak biały, spacja

Pętla for - listy

W jaki sposób wyświetlić liczby w jednej linii?

Można zmienić wartość parametru end na end="" #to jest znak biały, spacja

```
my_list = [1, 2, 3, 4, 5]
for element in my_list:
    print(element, end=" ")
```

Co zostanie wyświetlone?

1 2 3 4 5

Co się stanie jak ustawię parametr end="abc" ? #tam jest abc spacja ☺

1abc 2abc 3abc 4abc 5abc

Pętla for – listy

Jak wygląda rozbudowana pętla for?

```
list1=['wilk', 'ryś', 'kurczak','wieloryb']

for i, n in enumerate(list1):
    if n=="kurczak":
        print("index",i)
        print(n)
        print(list1[i-1])    #sprawdzenie elementu wcześniejszego
        print(list1[i+1])    #sprawdzenie elementu następnego
        print("tu kurczak, jestem na liście")
```

Co zostanie wyświetlone?

kurczak jesteś na liście?

index 2

kurczak

ryś

wieloryb

tu kurczak, jestem na liście

Pętla for - listy

Pętla for potrafi iterować po listach:

```
for x in ['wilk', "ryś", 'kurczak', 'wieloryb']:
    print(x)
    if x == "ryś":
        print("znaleziono zwierzę ", x)
```

wilk

ryś

znaleziono zwierzę ryś

kurczak

wieloryb

Zwróć uwagę na wcięcie w kodzie. Gdzie zaczyna i kończy się pętla a gdzie zaczyna i kończy się instrukcja warunkowa. Czy można powiedzieć, że instrukcja warunkowa znajduje się wewnątrz pętli?

Pętla for – listy składane

Co to są listy składane?

```
array=[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]

# array=[[1,2,3],
#        [4,5,6],
#        [7,8,9],
#        [10,11,12]]

array2=[row[1] for row in array]
array3=[row[2] for row in array]

print(array2)
print(array3)
```

Co zostanie wyświetlone?

```
[2, 5, 8, 11]
[3, 6, 9, 12]
```

Ten kod wykonuje dwie operacje na liście dwuwymiarowej (array) za pomocą składni list comprehension w Pythonie. Lista dwuwymiarowa array zawiera podlisty, a następnie są tworzone dwie nowe listy (array2 i array3), z których każda zawiera wartości z określonego indeksu dla każdej podlisty w array

Pętla for – łańcuchy znaków

Jak przebiega proces iterowania po łańcuchach znaków?

```
for letter in 'Python':
    if letter == 'h':
        pass
        print ("This is pass block")
    else:
        print ("Current Letter :",letter)
print ("Good bye!")
```

Pętla for – łańcuchy znaków

Pętla for współpracuje z instrukcją warunkową if.

Co zostanie wyświetlone?

```
for i in "lokomotywa":
    if i=="o":
        print("znaleziono literę o")
    else:
        print("nie znaleziono litery o")
```

```
nie znaleziono litery o
znaleziono literę o
nie znaleziono litery o
znaleziono literę o
nie znaleziono litery o
znaleziono literę o
nie znaleziono litery o
nie znaleziono litery o
nie znaleziono litery o
nie znaleziono litery o
```

Pętla for - zakresy

Pętla for jest jednym z podstawowych elementów języka Python, służącym do wykonywania pewnej sekwencji instrukcji wielokrotnie.

Składnia pętli for w Pythonie jest następująca:

```
zmienna_pomocnicza=wartość
for element in zakres(początek zakresu, koniec zakresu, skok)
    instrukcja do wykonania
    instrukcja do wykonania
```

Pętla for

Gdzie zmienna to zmienna iteracyjna, która przyjmuje kolejne wartości z zakresu podanego w zakresie. Pętla wykonuje instrukcje zawarte w jej ciele tak długo, aż zmienna iteracyjna osiągnie wartość końcową zakresu.

Przykładowo, aby wyświetlić na ekranie wszystkie liczby od 1 do 10, można użyć pętli for w następujący sposób:

```
for i in range(1,11):
    print(i)
```

Pętla „for” w języku Python będzie wykonywana określoną ilość razy aż do momentu, w którym zakres liczbowy się skończy. Na początku pętli zmienna sterująca (licznikowa) jest ustawiana na wartość początkową, a następnie przy każdym obrocie pętli jej wartość jest zwiększana o jeden, aż do osiągnięcia górnego limitu.

W powyższym przykładzie range(1, 11) tworzy sekwencję liczb od 1 do 10, zmienna i jest zmienną iteracyjną, która przyjmuje kolejno wartości z tej sekwencji. **Uwaga na przedział lewostronnie zamknięty i prawostronnie otwarty <1,11)**

Pętla for - zastosowania

Zadanie:

Oblicz sumę liczb z przedziału od 1 do 11.

```
sum=0
for i in range(1,12):
    sum=sum+i

print(sum)
```

Zauważ, że pętla for nie zaczyna się od razu od for. Wielokrotnie w kodzie trzeba przygotować zmienną, która będzie wykorzystywana w pętli.

Pętla for - zakresy

Zadanie:

Wygeneruj liczby od 350 do 320. Wygenerowane liczby powinny być podzielne przez dwa.

```
for i in range(350,320,-2):
    print(i)
```

Co zostanie wyświetlone?

350
348
346
344
342
340
338
336
334
332
330
328
326
324
322

Pętla for

Instrukcja **break** w Pythonie służy do przerwania dalszego wykonywania pętli (np. **for**, **while**) i wyjścia z niej. Jest ona często używana w połączeniu z instrukcją warunkową (np. **if**), aby sprawdzić jakiś warunek i w zależności od jego spełnienia lub niespełnienia przerwać dalsze wykonywanie pętli.

Przykład użycia instrukcji **break** w pętli **for**:

```
for i in range(1,11):
    if i==5:
        break
    print(i)
```

1
2
3
4
5

W powyższym przykładzie pętla **for** wykonuje się tylko 4 razy i przerywana jest po wypisaniu liczby 4, ponieważ w warunku **if** sprawdzane jest, czy **i** jest równe 5, a jeśli tak, to instrukcja **break** przerywa dalsze wykonywanie pętli.

Pętla for

Instrukcja **break** może być również używana wewnątrz zagnieżdżonych pętli, aby przerwać tylko wewnętrzną pętlę lub też zewnętrzną i wewnętrzną jednocześnie. W takim przypadku należy użyć słowa kluczowego **break** w odpowiedniej pętli.

Przykład użycia instrukcji **break** w zagnieżdżonych pętlach:

```
for i in range(1,3):
    for j in range(1,3):
        if i==2 and j==2:
            break
    print(i,j)
```

1 1
1 2
2 1
2 2

W powyższym przykładzie zewnętrzna pętla **for i in range(1, 3)** wykonuje się 2 razy, a wewnętrzna pętla **for j in range(1, 3)** również wykonuje się 2 razy. Jednak w warunku **if** sprawdzane jest, czy **i** jest równe 2 oraz **j** jest równe 2, a jeśli tak, to instrukcja **break** przerywa dalsze wykonywanie wewnętrznej pętli.

Pętla for - continue

Instrukcja **continue** może być również używana w pętli **for**. Na przykład:

```
1 for x in range(5):
2     if x%2==0:
3         continue
4     print(x)
```

W powyższym przykładzie pętla **for** iteruje przez liczby od 0 do 4 i wyświetla na ekranie tylko nieparzyste wartości.

Czy można iterować po obiekcie za pomocą pętli while?

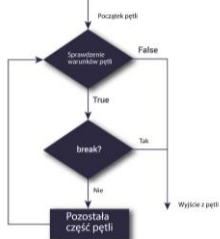
Instrukcja „continue” podobnie jak instrukcja „break” jest związana z pętlami i podobnie jak instrukcję „break” można ją stosować zarówno w przypadku pętli **for** lub **while**. Instrukcja „continue” powoduje przerwanie wykonania bieżącego kroku pętli i przejście do następnego kroku. Instrukcja „continue” działa tylko na pętlę w której się bezpośrednio znajduje - nie da się spowodować przerwania wykonania kroku pętli zewnętrznej.

```
Przykład pętli „while” z „continue” i listą.
1 i = 0
2 liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 while i < len(liczby):
4     if liczby[i] % 2 == 0:
5         i += 1
6         continue
7     print(liczby[i])
8     i += 1
```

W powyższym przykładzie pętla **while** powtarza się przez elementy listy i sprawdza czy element jest parzysty. Jeśli tak, komenda **continue** przeskakuje do kolejnej iteracji i nie wykonuje komendy **print**. Dlatego, wynikiem jest wyświetlenie wszystkich liczb nieparzystych z listy.

Pętla for

Jak wygląda przepływ informacji w pętli **for** z zastosowaniem instrukcji **break**?



Instrukcja break w pętli- gdzie umieszczać instrukcję break?

Instrukcja break przerywa pętlę, w której została zadeklarowana. Kontrola programu jest następnie przekazywana do instrukcji, która następuje po ciele pętli.

Jeśli instrukcja break znajduje się wewnątrz pętli zagnieżdżonej (pętla w pętli), to pętla wewnętrzna zostaje przerwana.

Pętla for

Funkcja len podaje długość listy łańcucha. Możemy jej użyć w pętli, gdy nie znamy długości listy.

Zadanie

Określ jaki jest wynik działania programu:

```
a = ['Marek', 'Ela', 'Adam', 'Jurek']
for i in range(len(a)):
    print(i, a[i])
```

Zadanie Zmodyfikuj powyższy program w ten sposób, aby wyświetlał imiona z listy a oraz z ilu liter się składają.

Instrukcja continue i break w pętlach

Pętle nie zawsze muszą być realizowane od początku do końca. Czasami, w zależności od warunków, może być konieczne pominięcie niektórych kroków pętli lub przedwczesne zakończenie pętli. Do tego właśnie służą instrukcje break i continue.

Break to instrukcja wcześniejszego zakończenia pętli.

Continue to instrukcja pominięcia reszty pętli i przejścia do następnego kroku pętli. W tym przypadku pętla nie jest zakończona.

Instrukcje break i continue sterują pętlą.

Pętle wykonują blok, gdy tak długo jak warunek pętli jest prawdziwy. Czasami potrzebujemy przerwać wykonywanie całej pętli bez sprawdzania warunku.

W takich przypadkach należy zastosować instrukcję break lub continue.

Pętla for - continue

Instrukcja **continue** w Pythonie jest używana wewnątrz pętli (np. **while**, **for**, itd.), aby przerwać bieżącą iterację i przejść do następnej. Po wykonaniu instrukcji **continue** pętla wraca do swojego warunku kontrolnego i rozpoczyna kolejną iterację.

Oto przykład użycia instrukcji **continue** w pętli **while**:

```
1 x=0
2
3 while x<5:
4     x+=1
5     if x%2 == 0:
6         continue
7     print(x)
```

W powyższym przykładzie pętla **while** wykonywana jest do momentu, aż zmienna **x** nie osiągnie wartości 5. W każdej iteracji pętli sprawdzany jest warunek **x % 2 == 0**, a jeśli jest on prawdziwy, instrukcja **continue** jest wykonywana, co oznacza, że reszta kodu w pętli jest pomijana i rozpoczyna się kolejna iteracja. W rezultacie na ekranie wyświetlone zostają tylko nieparzyste wartości zmiennej **x**.

Pętla for - continue

Polecenie **continue** działa analogicznie jak **break**, tylko że pętla nie jest przerywana, a jedynie pomijany jest kod po **continue** i pętla dalej kontynuuje działanie.

Zadanie

Ustalić jaki jest wynik działania programu. Co wykonuje ten program?

```
x=[2,-1,3,-2,9]
for i in x:
    if i<0:
        continue
    print(i**0.5)
```

Zadanie

a) Używając m.in. poleceń: **range**, **continue** i % napisz program wyznaczający kwadraty wszystkich liczb naturalnych od 0 do 100 niepodzielnych przez 6.

b) Używając m.in. poleceń: **range**, **continue**, % oraz not napisz program wyznaczający sumę wszystkich liczb naturalnych od 1 do 1000 podzielnych przez 25.

Zagnieżdzenie pętli

Zagnieżdżone pętli działają w ten sposób, że Na każdy obrót zewnętrznej pętli przypada pełny cykl obrotów wewnętrznej pętli np.:

```
for i in range(0,10):
    print('petla zewnetrzna: ', i)
    for j in range(0,10):
        print('petla wewnetrzna: ', j)
```

```
petla zewnetrzna: 0
petla wewnetrzna: 0
petla wewnetrzna: 1
petla wewnetrzna: 2
petla wewnetrzna: 3
petla wewnetrzna: 4
petla wewnetrzna: 5
petla wewnetrzna: 6
petla wewnetrzna: 7
petla wewnetrzna: 8
petla wewnetrzna: 9
petla zewnetrzna: 1
```

Pętla for – iterowanie po listach dwuwymiarowych

```
list = [[80, 60, 70], [10, 20, 30]]
```

```
for i in list:
    for j in i:
        print(j)
```

```
80
60
70
10
20
30
```

Pętla zewnętrzna obsługuje listę [80,60,70] natomiast pętla wewnętrzna obsługuje elementy tej listy.

Pętla for

Zadanie:

Oblicz silnię dla n elementów. Użyj do tego pętli for.

```
n = input("Podaj dla ilu liczb chcesz obliczyć silnię")

def silnia(n):
    wynik = 1
    for i in range(1, n + 1):
        wynik *= i
    return wynik

print("Silnia z 5:", silnia(5))
```

Pętla for

Zadanie:

Znajdź liczby pierwsze

z przedziału liczb od 1 do 50:

```
def czy_pierwsza(num):
    if num > 1:
        for i in range(2, int(num/2) + 1):
            if (num % i) == 0:
                return False
        else:
            return True
    else:
        return False

print("Liczby pierwsze od 1 do 50:")
for i in range(1, 51):
    if czy_pierwsza(i):
        print(i)
```

Pętla for

Zadanie:

Znajdź liczbę która określa ile razy litera a znajduje się w wyrazie.

```

napis = "To jest przykładowy napis"
liczba_a = 0
for litera in napis:
    if litera == "a":
        liczba_a += 1
print("Liczba liter 'a' w napisie:", liczba_a)

```

Pętla for

Zadanie:

Znajdź kwadraty liczb z zakresu od 1 do 5. Użyj do tego listy składanej.

```

kwadraty = [i**2 for i in range(1, 6)]
print("Lista kwadratów liczb od 1 do 5:", kwadraty)

```

Pętla for

Zadanie:

Zsumuj liczby które znajdują się w liście.

```

lista = [10, 20, 30, 40, 50]
suma = 0
for element in lista:
    suma += element
print("Suma elementów listy:", suma)

```

Pętla for**Zadanie:**

Przekonwertuj małe litery w łańcuchu znaków na duże litery.

```

napis = "to jest przykładowy napis"
napis_duże_litery = ""
for litera in napis:
    napis_duże_litery += litera.upper()
print("Napis z dużymi literami:",
      napis_duże_litery)

```

Pętla for – alternatywne spojrzenie na map function

Czy trzeba za każdym razem budować konstrukcję pętli for jeśli chcemy coś przeiterować? Nie!

Funkcja map w Pythonie to wbudowana funkcja, która umożliwia zastosowanie określonej funkcji do każdego elementu w sekwencji (takiej jak lista, tuple czy zbiór) i zwrócenie nowej listy zawierającej wyniki tej funkcji.

Mapowanie jest jednym ze sposobów zastosowania funkcji do wielu elementów jednocześnie, co może pomóc w zwiększeniu czytelności i efektywności kodu.

```
map(function, iterable, ...)
```

function: To jest funkcja, którą chcemy zastosować do każdego elementu sekwencji.
iterable: To jest sekwencja, której elementy chcemy przekształcić za pomocą funkcji.

Pętla for**Zadanie:**

Przeiteruj listę podnosząc jej elementy do kwadratu. Nie używaj pętli for.

```

def square(x):
    return x**2
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)

print(list(squared_numbers))    # Output: [1, 4, 9, 16, 25]

```

W tym przykładzie, funkcja square jest zastosowana do każdego elementu listy numbers za pomocą funkcji map. Wynikiem jest obiekt map, który można przekształcić na listę (za pomocą list()), aby uzyskać ostateczny rezultat.

Funkcja map jest przydatna, gdy chcemy zastosować tę samą operację do każdego elementu sekwencji, bez potrzeby używania pętli.

Zadanie z zastosowaniem pętli for

Zadanie:

Napisać program, który wypisze:

- a) kwadraty wszystkich liczb całkowitych od 0 do 20,
- b) sześciiany wszystkich liczb całkowitych od 10 do 20,
- c) odwrotności wszystkich parzystych liczb całkowitych od 16 do 6 (w podanej kolejności)

Omów poniższy kod – „ale o co chodzi?”

```
def podwojenie(x):
    return x * 2

liczby = [1, 2, 3, 4, 5]
wyniki = map(podwojenie, liczby)
wyniki_lista = list(wyniki)
print(wyniki_lista)
```

Wyższa Szkoła Przedsiębiorczości i Administracji



Programowanie

Pętla while
Obsługa plików

Prowadzący: dr inż. Sylwester Korga

Własność materiałów edukacyjnych: dr inż. Sylwester Korga

O czym należy pamiętać aby poprawnie stosować pętlę while w kodzie?

1. Składnia while w Pythonie:

Omówienie, jak wygląda składnia pętli while w Pythonie, czyli jak deklarować pętlę, używając słowa kluczowego while, warunku oraz dwukrotnego dwukropka :.

2. Warunek pętli: Wyjaśnienie, co to jest warunek pętli, czyli wyrażenie logiczne, które jest sprawdzane na początku każdego przejścia przez pętlę. Pętla działa tak długo, jak warunek jest spełniony.

3. Ciało pętli: Opis, co zawiera ciało pętli, czyli blok kodu, który jest wykonywany, dopóki warunek jest spełniony. W tym miejscu znajdują się instrukcje lub operacje wykonywane w każdym przejściu pętli.

4. Inicjacja zmiennych: Jeśli w pętli używasz zmiennych, to warto omówić, jakie zmienne są inicjowane przed pętlą i jak zmienne te ewentualnie zmieniają się w trakcie działania pętli.

5. Warunek zakończenia: Wyjaśnienie, kiedy pętla zostaje zakończona, czyli co powoduje, że warunek staje się fałszywy, a pętla przerywa działanie.

O czym należy pamiętać aby poprawnie stosować pętlę while w kodzie?

6. Instrukcje kontrolne: Omówienie instrukcji kontrolnych, takich jak break (przerwij) i continue (kontynuuj), które pozwalają na bardziej zaawansowaną kontrolę nad działaniem pętli.

7. Bezpieczeństwo pętli: Warto podkreślić, że pętle while mogą prowadzić do nieskończonych pętli, jeśli warunek nie zostanie spełniony. Dlatego istotne jest, aby zadbać o bezpieczeństwo i umiejętnie zdefiniować warunek zakończenia.

8. Testowanie pętli: Dobrym pomysłem jest zaprezentowanie kilku przykładów zastosowania pętli while w Pythonie, aby zilustrować różne scenariusze i techniki.

9. Optymalizacja i dobre praktyki: Omówienie potencjalnych optymalizacji kodu związanego z pętlami while, a także podanie dobrych praktyk programistycznych, takich jak czytelność kodu i komentarze.

10. Zastosowania pętli while: Przedstawienie różnych zastosowań pętli while w rzeczywistych problemach, aby pokazać, jak są one przydatne w praktyce.

Pętla while - definicja

Pętla „while” jest wykonywana tak długo dopóki określony warunek będzie prawdziwy. Warunek po każdym wykonaniu jest ponownie sprawdzany i jeśli jest prawdziwy zwraca wartość „True”, kod w bloku jest wykonywany. Jeżeli warunek jest fałszywy, wtedy przyjmie wartość „False”, blok kodu wewnątrz pętli się nie wykona.

Pętla while jest często używana do wykonywania operacji w pętli dopóki nie zostanie spełniony określony warunek, takie jak pobieranie danych od użytkownika lub dopóki nie zostanie podana poprawna wartość.

```
#pętla while odliczająca do 10
```

```
count=0
```

```
while count<10:
    print(count)
    count += 1 # count=count+1
```

While – schemat działania

Pętla while działa jeśli warunek jest spełniony.

Co to znaczy, że warunek jest spełniony?

To znaczy, że jego wynik posiada wartość logiczną TRUE.

Ciała pętli wykonywane jest tyle razy ile razy wynik warunku logicznego jest prawdziwy.

Jeśli warunek nie jest spełniony to ciało pętli się nie wykona.



Charakterystyka pętli while

Pętla while jest pętlą kontrolowaną warunkowo w Pythonie. Działa w następujący sposób:

najpierw sprawdzany jest warunek, a jeśli jest on prawdziwy, wykonywana jest pętla. Gdy warunek nie jest już spełniony, pętla jest zakończona.

Oto przykład pętli while w Pythonie:

```
count=0

while count<5:
    print(count)
    count += 1
```

W powyższym przykładzie pętla while wykonywana jest do momentu, aż zmienna x (linijka nr1;której sami przypisujemy wartość) nie osiągnie wartości 5. W każdej iteracji pętli zmienna x zwiększana jest o 1, dzięki czemu warunek $x < 5$ ostatecznie przestaje być spełniony i pętla zostaje zakończona.

Przed napisaniem pętli przygotuj zmienną lub zmienne

Uwaga! Zaczynając pętla while nie zaczyna się od „while”. W wielu przypadkach do prawidłowego działania pętli potrzebna jest zmienna ustalona przed kodem pętli. Spójrz na poniższy przykład:

```
#pętla while odliczająca do 10
count=0

while True:
    print(count)
    count += 1
    if count>=10:
        break
```

W pętlach bardzo często wykorzystuje się inkrementację oraz dekrementację o różnym zapisie:
 $x=x+1$ to jest to samo co $x+=1$
 $y=y-1$ to jest to samo co $y-=1$

W powyższym przykładzie pętla while jest nieskończona, ponieważ warunek jest zawsze prawdziwy (True). Instrukcja break służy do przerywania pętli, gdy zmienna x osiągnie wartość 5 lub większą.

Przykład zastosowania pętli while do autoryzacji hasła

Zadanie:

Napisz program, który uruchamia się po podaniu prawidłowego hasła. Hasło powinno być wyświetlane znakami typu *.

```
import pwinput

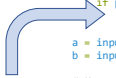
while True:
    password1 = pwinput.pwinput("Podaj hasło aby otworzyć program: ")
    if password1 == "abcd":
        break # Jeśli hasło jest poprawne, przerwij pętlę

a = input("Podaj pierwszy składnik sumy: ")
b = input("Podaj drugi składnik sumy: ")

# Konwertuj dane wejściowe na liczby (jeśli są liczbami)
try:
    a = float(a)
    b = float(b)
except ValueError:
    print("Błąd: Wprowadzone dane nie są liczbami.")
    exit()

wynik = a + b
print("Wynik dodawania:", wynik)
```

Zwróć uwagę, że instrukcja break przerywa działanie zarówno instrukcji if jak również pętli while.



Pętla while jako generator liczb

Pętla while może służyć jako generator liczb. Liczby te mogą być określone oddzielnie lub mogą być wybierane w sposób losowy.

Wykorzystanie pętli while do generowania liczb w podanym zakresie:

```
liczba = 10

while liczba <= 20:
    print(liczba)
    liczba += 1
```

Wykorzystanie pętli while do generowania liczb losowych:

```
import random

while True:
    random_number = random.randint(1, 10)
    if random_number == 8:
        print("wylosowano liczbę 8")
        break
```

Nietypowe zastosowanie pętli while

Przykład zastosowania pętli while, która sprawdza czy jest dostępne połączenie z Internetem.

```
import socket

while True:
    try:
        socket.create_connection(("www.wspa.pl", 80))
        print("Połączenie jest dostępne")
        break
    except OSError:
        print("Brak połączenia z internetem")
```

W tym wyrażeniu, socket.create_connection to wywołanie funkcji, a argumentem jest krotka ("www.example.com", 80). Działa to tak, że tworzy połączenie z serwerem o adresie "www.example.com" na porcie 80.

Zastosowanie instrukcji break i continue w pętli while

Przykład zastosowania instrukcji break w pętli while:

Pętla while działa dopóki x jest mniejsze niż 5

```
x = 0
while x < 5:
    if x == 3:
        break # Przerwij pętlę, gdy x osiągnie wartość 3
    print(x)
    x += 1
```

Jak wyglądają konstrukcje pracy z plikami?

W języku Python można wyróżnić dwie konstrukcje wykorzystywane do pracy z plikami:

with open("plik.txt", 'a') as file:
 file.write("Nowa linia tekstu\n")
#nie zamykam pliku bo nie muszę

file = open("plik.txt", 'a')
file.write("Nowa linia
tekstu\n")
file.close()

Przykład wykorzystania konstrukcji „with open()”

Zadanie:

Wykonaj generator umożliwiający wygenerowanie liczb od 1 do 5. Wygenerowane liczby będą zapisane do pliku o nazwieplik.txt. Liczby są zapisane w taki sposób, że każda liczba znajduje się w osobnej linii.

```
with open("nazwa1.txt", "w+") as file:
    for e in range(1,6):
        e=str(e)
        file.write(e + "\n")
```

nazwa1.txt

```
1
2
3
4
5
```

Tryb zapisu pliku

Tryby otwierania plików (8 trybów)

	r	r+	w	w+	a	a+
Read from file	Yes	Yes	No	Yes	No	Yes
Write to file	No	Yes	Yes	Yes	Yes	Yes
Create file if not exists	No	No	Yes	Yes	Yes	Yes
Truncate file to zero length	No	No	Yes	Yes	No	No
Cursor position	Beginning	Beginning	Beginning	Beginning	End	End
Only reading	r					
Only writing, truncate	w					
Only writing, no truncate	a					
Reading and writing, truncate	w+					
Reading and writing, no truncate	r+					

Tryby otwierania plików (8 trybów)

	r	r+	w	w+	a	a+
Read from file	Yes	Yes	No	Yes	No	Yes
Write to file	No	Yes	Yes	Yes	Yes	Yes
Create file if not exists	No	No	Yes	Yes	Yes	Yes
Truncate file to zero length	No	No	Yes	Yes	No	No
Cursor position	Beginning	Beginning	Beginning	Beginning	End	End
Only reading	r					
Only writing, truncate	w					
Only writing, no truncate	a					
Reading and writing, truncate	w+					
Reading and writing, no truncate	r+					

"r" (read): Ten tryb jest używany do odczytywania danych z istniejącego pliku. Jeśli plik nie istnieje, to zostanie zgłoszony błąd.

"r+" (read and write): Ten tryb pozwala na odczytywanie i zapisywanie danych do istniejącego pliku. Jeśli plik nie istnieje, to zostanie zgłoszony błąd.

"a+" (append and read): Ten tryb pozwala na odczytywanie danych z istniejącego pliku oraz dodawanie nowych danych na jego końcu. Jeśli plik nie istnieje, to zostanie utworzony.

"a" (append): Ten tryb pozwala tylko na dodawanie nowych danych na końcu istniejącego pliku. Jeśli plik nie istnieje, to zostanie utworzony.

"w" (write): Ten tryb jest używany do zapisywania danych do pliku. Jeśli plik już istnieje, to zostanie nadpisany. Jeśli plik nie istnieje, to zostanie utworzony.

"w+" (write and read): Ten tryb pozwala na zapisywanie danych do pliku oraz odczytywanie danych z pliku. Jeśli plik już istnieje, to zostanie nadpisany. Jeśli plik nie istnieje, to zostanie utworzony.

Dodatkowe tryby pracy z plikami

"x" (exclusive creation): Ten tryb służy do tworzenia nowego pliku do zapisu. Jeśli plik o podanej nazwie już istnieje, to zostanie zgłoszony błąd `FileExistsError`. W przeciwnym razie zostanie utworzony nowy pusty plik.

"x+" (exclusive creation and reading): Ten tryb działa podobnie jak "x", ale dodatkowo umożliwia odczytywanie danych z nowo utworzonego pliku. Jeśli plik o podanej nazwie już istnieje, to zostanie zgłoszony błąd `FileExistsError`. W przeciwnym razie zostanie utworzony nowy pusty plik i będzie można go odczytać i zapisywać dane.

Przy wykorzystaniu trybu x lub x+ należy korzystać z konstrukcji try, except.

```
try:
    with open('nowy_plik.txt', 'x') as file:
        file.write('To jest nowy plik.')
except FileExistsError:
    print("Plik już istnieje.")
```

Wykorzystanie atrybutów funkcji open() jako trybów pracy:

Otwieranie pliku w trybie do odczytu

```
with open('plik.txt', 'r') as file:
    data = file.read()
```

Otwieranie pliku w trybie do zapisu

```
with open('plik.txt', 'w') as file:
    file.write('Nowa zawartość pliku')
```

Pamiętaj, że korzystanie z tych trybów otwierania plików wiąże się z pewnym ryzykiem, zwłaszcza w przypadku trybów zapisu, ponieważ nadpisanie lub usunięcie istniejących danych jest możliwe. Dlatego zawsze należy zachować ostrożność przy operacjach na plikach.

Jak odczytać dane z pliku przy pomocy pętli while?

#Otwieramyplik do odczytu

```
with open('plik.txt', 'r') asplik:
    linia =plik.readline() # Odcytujemy pierwszą linię i zapisujemy do zmiennej linia
```

```
while linia:
    # Przykładowa operacja na linii
    print(linia.strip()) # Usuwamy białe znaki z końca i początku linii i wyświetlamy
    linia =plik.readline() # Odcytujemy kolejną linię
```

Po zakończeniu pętli while, plik zostanie automatycznie zamknięty

W tym przykładzie otwieramy plik "plik.txt" do odczytu, a następnie używamy pętli while, aby odczytać i przetworzyć każdą linię pliku. Pętla while jest używana do odczytywania linii pliku w pętli do momentu, gdy osiągnię się koniec pliku. Po zakończeniu pętli plik jest automatycznie zamykany dzięki użyciu with open.

Powyższy przykład jest tylko jednym ze scenariuszy, w którym można połączyć pętlę while z obsługą plików. W zależności od potrzeb, można przetwarzać pliki w pętlach while, wykorzystując różne warunki i operacje na danych odczytanych z pliku.

Metody wykonywane na aliasie pliku

Na obiekcie (aliasie) file można wywołać wiele metod. Oto niektóre z nich.

write(str): Służy do zapisywania tekstu (łańcuchów znaków) do pliku.

writelines(lines): Pozwala na zapisanie listy linii (łańcuchów znaków) jako kolejnych linii w pliku.

seek(offset, whence): Umożliwia przesunięcie wskaźnika pozycji w pliku. Przydatne, gdy chcesz zapisywać dane w określonym miejscu pliku.

truncate(size): Pozwala na przycięcie pliku do określonej wielkości. Jeśli zostanie pominięty argument size, to przycina do obecnej pozycji wskaźnika.

flush(): Wywołanie tej metody sprawi, że wszystkie buforowane dane zostaną zapisane na dysku.

file.read() Odczytuje całą zawartość pliku
file.read(50) Odczytuje zawartość 50 znaków

close(): Służy do ręcznego zamknięcia pliku. Chociaż w przypadku użycia with, plik zostanie automatycznie zamknięty po opuszczeniu bloku with, to można także zamknąć go ręcznie.

Manipulowanie plikami typu pdf

Współpraca z plikami pdf odbywa się przy wykorzystaniu podobnej konstrukcji programistycznej jak przy plikach txt.

```
import PyPDF2
```

```
# Otwarcie pliku PDF w trybie odczytu
with open("plik.pdf", "rb") as pdf_file:
    pdf_reader = PyPDF2.PdfFileReader(pdf_file)
```

```
# Pobranie liczby stron w pliku PDF
liczba_stron = pdf_reader.numPages
print(f"Liczba stron: {liczba_stron}")
```

```
# Odczyt zawartości każdej strony
for numer_strony in range(liczba_stron):
    strona = pdf_reader.getPage(numer_strony)
    tekst = strona.extractText()
    print(f"Zawartość strony {numer_strony}: \n{tekst}") przy plikach txt.
```

Tryb "rb" jest używany do odczytywania plików binarnych, w których dane są reprezentowane w formie ciągów bajtów bez interpretacji kodowania znaków. Jest to odpowiednie do plików binarnych, takich jak pliki PDF, obrazy (JPEG, PNG itp.), pliki dźwiękowe (MP3, WAV itp.), a także innych plików, które nie są w formacie tekstowym

Wyższa Szkoła Przedsiębiorczości i Administracji



Wyższa Szkoła
Przedsiębiorczości
i Administracji

Programowanie

Listy, krotki, tablice, słowniki, zbiory

Prowadzący: dr inż. Sylwester Korga

Własność materiałów edukacyjnych: dr inż. Sylwester Korga

Co to jest lista w programowaniu?

Listy to jedno z podstawowych struktur w językach programowania.

Z definicji są one zmiennymi zawierającymi uporządkowany zbiór elementów.

Lista jest to zmienna zawierająca uporządkowany zbiór elementów. Podobnie jak łańcuchy znaków (strings) tworzy ciąg elementów. W odróżnieniu od łańcuchów znaków, listy są modyfikowalne, czyli można dodawać i zmieniać poszczególne elementy danej listy.

Listę tworzone są za pomocą nawiasów kwadratowych [], gdzie pomiędzy nimi wypisane są jej elementy rozdzielone przecinkami.

```
# utworzenie listy pustej
pusta_lista = []
# utworzenie listy zawierającej elementy
lista_z_elementami = [1, 5, 9, 'tekst']
```

Czy lista to samo co tablica?

Tablica odnosi się do struktury danych, która pozwala przechowywać zbiór elementów o identycznym typie danych pod jednym wspólnym identyfikatorem. Tablice są używane do organizowania danych w sposób, który umożliwia łatwe odwoływanie się do poszczególnych elementów.

W wielu językach programowania tablice są jednym z podstawowych typów danych.

Elementy w tablicy są przechowywane pod kolejnymi indeksami (numerami), zazwyczaj zaczynając od 0. Dzięki indeksom, można łatwo uzyskać dostęp do konkretnego elementu w tablicy.

Lista odnosi się do struktury danych, która przechowuje kolekcję elementów w określonej kolejności. Elementy w liście mogą być różnych typów danych, w zależności od języka programowania. Listy są bardzo użyteczne, ponieważ umożliwiają przechowywanie wielu wartości w jednej zmiennej i manipulację nimi w trakcie wykonywania programu.

```
#zdefiniowanie i wyświetlenie przykładowe Listy
Lista_nr5 = [11, 4.6, "element typu tekstowego", True]
print(lista_nr5)
```

Czy lista to dynamiczna tablica?

W Pythonie, "lista" to dynamiczna tablica, ponieważ wewnętrznie jest to implementowane jako tablica, która automatycznie się dostosowuje do rozmiaru w miarę dodawania i usuwania elementów. Możesz tworzyć listy, dodawać do nich elementy, usuwać elementy, a Python zarządza alokacją pamięci i rozszerzaniem/zmniejszaniem tablicy za ciebie.

W innych językach programowania, termin "dynamiczna tablica" może być używany w odniesieniu do struktur danych, które oferują podobne funkcjonalności jak listy w Pythonie, ale terminologia może być różna (na przykład "ArrayList" w Javie lub "vector" w C++). W tych językach dynamiczna tablica może być bardziej jawnie zarządzana niż w Pythonie, gdzie lista ukrywa wiele szczegółów implementacyjnych.

W Pythonie, w przeciwieństwie do niektórych innych języków programowania, nie istnieje dokładnie zdefiniowany typ "tablica statyczna" tak, jak to jest w językach takich jak C czy Java. Python oferuje bardziej elastyczne struktury danych, takie jak listy.

Operacje na listach

W jaki sposób przekonwertować ciąg znaków na listę?

Trzeba użyć metody split() oraz określić separator.

Metoda split() w języku Python jest używana do dzielenia ciągu znaków na podstawie określonego separatora. Przykład użycia metody split():

```
# Przykładowy ciąg znaków
ciag_znakow = "Jeden,Dwa,Trzy,Cztery,Pięć"

# Użycie metody split() do podzielenia ciągu na podstawie przecinka (",")
podzielony_ciag = ciag_znakow.split(',')

# Wyświetlenie wyniku
print("Pierwotny ciąg znaków:", ciag_znakow)
print("Podzielony ciąg:", podzielony_ciag)
```

```
# Pierwotny ciąg znaków: Jeden,Dwa,Trzy,Cztery,Pięć
# Podzielony ciąg: ['Jeden', 'Dwa', 'Trzy', 'Cztery', 'Pięć']
```


Operacje na listach- uwaga na przypisywanie listy do listy

Jeśli utworzysz kopię listy za pomocą przypisania, np. `kopia_listy = lista_oryginalna`, to obie zmienne (`kopia_listy` i `lista_oryginalna`) będą wskazywały na ten sam obiekt listy. W takim przypadku, jeżeli zmodyfikujesz oryginalną listę, zmiany będą odzwierciedlone w kopii i vice versa.

```
# Oryginalna lista
lista_oryginalna = [1, 2, 3, 4]

# Utworzenie referencji listy a nie jej kopii !!!
referencja_listy = lista_oryginalna

# Modyfikacja oryginalnej listy
lista_oryginalna.append(5)

# Wydruk obu list
print("Oryginalna lista:", lista_oryginalna)
print("Referencja listy:", referencja_listy)
```

Oryginalna lista: [1, 2, 3, 4, 5]
referencja listy: [1, 2, 3, 4, 5]

Operacje na listach- uwaga na przypisywanie listy do listy

Aby można było wykonywać operacje na nowej liście ale bez zmian na oryginalnej liście trzeba zrobić kopię.

```
# Oryginalna lista
lista_oryginalna = [1, 2, 3, 4]

# Utworzenie kopii listy
kopia_listy = lista_oryginalna.copy()

# Modyfikacja oryginalnej listy
kopia_listy.append(5)

# Wydruk obu list
print("Oryginalna lista:", lista_oryginalna)
print("Kopia listy:", kopia_listy)
```

Tworzenie kopii a nie referencji.
Można też i tak:
`kopia_listy = lista_oryginalna[:]`

Jeśli zrobisz operację na kopii to oryginał też zmieni.
Jeśli zrobisz operację na oryginale to kopię też zmieni. Tak działa referencja.

Oryginalna lista: [1, 2, 3, 4]
Kopia listy: [1, 2, 3, 4, 5]

Operacje na listach

Operacje wykonywane na obiekcie listy:

```
# utworzenie nowej listy
liczby_lista = ['jeden', 'dwa', 'trzy']
print(liczby_lista)          ['jeden', 'dwa', 'trzy']

# usunięcie elementu z listy
liczby_lista.remove('dwa')
print(liczby_lista)          ['jeden', 'trzy', 'cztery']

# dodanie elementu do listy
liczby_lista.append('cztery')
print(liczby_lista)          ['jeden', 'dwa', 'trzy', 'cztery']
```

Jak przypisywać listy i metody do nowej zmiennej?

Nieprawidłowo:

```
lista_a = [1, 2, 3]
lista_b = lista_a.append(4) # Modyfikacja oryginalnej listy

# Nowa zmienna wskazuje na zaktualizowaną listę
print(lista_b)

NONE
```

Prawidłowo:

```
lista_a = [1, 2, 3]
lista_a.append(4) # Modyfikacja oryginalnej listy

lista_b = lista_a # Nowa zmienna wskazuje na zaktualizowaną listę
print(lista_b)

[1, 2, 3, 4]
```

Dodawanie elementów do listy

W języku Python metoda insert() jest używana do wstawiania elementu na określoną pozycję w liście. Przykład użycia metody insert():

```
# Oryginalna lista
lista = [1, 2, 3, 4, 5]

# Wstawienie elementu na drugą pozycję (indeks 1)
lista.insert(1, 10) ← lista.insert(index, element)

# Wypdruk zaktualizowanej listy
print(lista)

[1, 10, 2, 3, 4, 5]
```

Operacje na listach

Metoda del jest używana do usunięcia elementu lub fragmentu listy na podstawie indeksu lub zakresu indeksów.

```
# Oryginalna lista
lista = [1, 2, 3, 4, 5]

# Usunięcie elementu o indeksie 2
del lista[2] ← To jest index a nie element

# Wypdruk zaktualizowanej listy
print(lista)

[1, 2, 4, 5]
```

```
# Oryginalna lista
lista = [1, 2, 3, 4, 5]

# Usunięcie elementów od indeksu 1 do indeksu 3 (bez 3)
del lista[1:3] ← To jest zakres indeksów.

# Wypdruk zaktualizowanej listy
print(lista)

[1, 4, 5]
```

Operacje na listach

Metoda pop usuwa element z listy na podstawie indeksu i zwraca ten element.

```
# Oryginalna lista
lista = [1, 2, 3, 4, 5]

# Usunięcie elementu o indeksie 2 i zwrócenie go
usuniety_element = lista.pop(2)

# Wydruk zaktualizowanej listy i usuniętego elementu
print("Zaktualizowana lista:", lista)
print("Usunięty element:", usuniety_element)

Zaktualizowana lista: [1, 2, 4, 5]
Usunięty element: 3
```

Przeszukiwanie list

Funkcje max(), min() działają nie tylko na liczbach ale również i na innych typach. Typów jednak nie można mieszać. Jeśli chodzi o napisy (stringi) to określenie wartości maksymalnej (minimalnej) opiera się na kolejności znaków w tablicy ASCII. W języku Python funkcje max() i min() są wbudowanymi funkcjami służącymi do znalezienia maksymalnej i minimalnej wartości w danym zbiorze elementów. Te funkcje mogą być używane na różnych rodzajach danych, takich jak listy, krotki czy sekwencje liczb.

```
# Na pojedynczych argumentach
max_value = max(3, 8, 1, 6, 2)
print("Największa wartość:", max_value)    Największa wartość: 8

# Na liście
lista = [5, 10, 3, 8, 2]
max_value = max(lista)
print("Największa wartość w liście:", max_value)    Największa wartość w liście: 10
```

Przeszukiwanie list

Funkcja min() zwraca najmniejszą wartość spośród przekazanych jej argumentów lub najmniejszą wartość w danym zbiorze elementów.

```
# Na pojedynczych argumentach
min_value = min(3, 8, 1, 6, 2)
print("Najmniejsza wartość:", min_value)

Najmniejsza wartość: 1

# Na liście
lista = [5, 10, 3, 8, 2]
min_value = min(lista)
print("Najmniejsza wartość w liście:", min_value)

Najmniejsza wartość w liście: 2
```

Operacje na listach

Operacje wykonywane na obiekcie listy:

#sprawdzenie czy dany element jest w liście

```
lista4 = ['a', 'b', 'c', 'd', 'a']
'a' in lista4
```

True

#sprawdzenie indeksu elementu 'b'

```
lista4 = ['a', 'b', 'c', 'd', 'a']
lista4.index('b')
```

1

#sprawdzenie ile razy 'a' występuje w liście

```
lista4 = ['a', 'b', 'c', 'd', 'a']
lista4.count('a')
```

2

Operacje na listach – sortowanie, sort() i sorted()

Czym się różni metoda sort() od metody sorted()?

Sortowanie listy oryginalnej, operacja przeprowadzona na oryginale.

```
lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

Sortowanie w miejscu (modyfikuje oryginalną listę).

lista.sort()

print("Posortowana lista w miejscu:", lista)

Metoda sort() nie zwraca oryginalnej listy.
Jak przypiszemy do niej zmienną to zmienna
otrzyma wartość None**Sortowanie kopii bez modyfikacji oryginału.**

```
lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

Uzyskanie posortowanej kopii listy (oryginalna lista

pozostaje bez zmian)

posortowana_lista = sorted(lista)

print("Oryginalna lista:", lista)

print("Posortowana lista (kopia):", posortowana_lista)

Metoda sorted() zwraca posortowaną

kopię i jednocześnie zwraca oryginał.

Operacje na listach – odwracanie zawartości listy

Na jakiej zasadzie działa metoda reverse()?

#funkcja reverse() odwraca kolejność

elementów

moja_lista5 = ['a', 'c', 'b', 'g', 'e']

['e', 'g', 'b', 'c', 'a']

moja_lista5.reverse()

print(moja_lista5)

Odwrócić listę można za pomocą metody reverse() lub za pomocą własnego kodu:

```
L=[1,2,3,4,5,6,7,8,9,10]
i = 0
j = len(L)- 1
while i < j:
    L[i], L[j] = L[j], L[i]
    i += 1
    j -= 1

print(L)
```

Listy zagnieżdżone

Co to są listy zagnieżdżone

```
#tworzenie tablicy
lista001 = ['a', 'b', 'c']
lista002 = ['d', 'e', 'f']
lista003 = ['g', 'h', 'i']
lista=[lista001, lista002, lista003]
print(lista)

#wyświetlenie pierwszego wiersza tablicy
lista001[0]

#wyświetlenie pierwszej kolumny tablicy
kolumna1 = [wiersz[0] for wiersz in lista]
print(kolumna1)
```

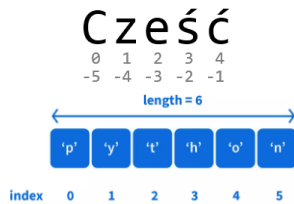
Output>
[['a', 'b', 'c'],
['d', 'e', 'f'],
['g', 'h', 'i']]

Output>
['a', 'b', 'c']

Output>
['a', 'd', 'g']

Co to jest numer indeksu w liście?

Podobnie jak przy napisach (strings) można się odwołać do wybranego elementu listy. Elementom listy przyporządkowane są konkretne numery indeksu. Pierwszy element listy ma zawsze numer 0, drugi 1 itd.



Operacje na listach

Operacje, które wykonywane są na liście w Pythonie przy użyciu nawiasów kwadratowych, to operacje indeksowania i slicing.

```
#Utworzenie listy z elementami
list1 = ['jeden', 'dwa', 'trzy', 'cztery']

#Wybór pierwszego elementu z listy
list1[0]

#Wybór ostatniego elementu z listy
list1[-1]

#wybór elementów od indeksu nr 2
list1[2:]
```

'jeden'

'cztery'

['trzy', 'cztery']

Dodawanie elementu do listy

W jaki sposób można pobrać elementy z listy? Co to jest Slicing?

Slicing to operacja używana w wielu językach programowania do wydobywania fragmentu danych (np. listy, tablicy, ciągu znaków) poprzez określenie zakresu elementów, które chcemy uzyskać. Jest to sposób na uzyskanie podzbioru danych z większej kolekcji.

```
nowa_lista = [1,2,3,4,5]

# Wywołanie elementu listy za pomocą indeksu
moja_lista[1] # zwróci wartość drugiego elementu listy, czyli tutaj liczba 2
moja_lista[3] # = 4
moja_lista[10] # Indeks poza zakresem listy, Python zwróci tutaj błąd: 'IndexError: list index out of range'

# Możemy też wywołać element listy licząc ostatniego elementu
moja_lista[-1] # = 5
moja_lista[-2] # = 4

# Wycinanie ('slicing') wybranego ciągu elementów z listy
moja_lista[1:] # Dostaniemy tutaj listę składającą się z czterech elementów [2,3,4,5]
moja_lista[1:3] # Fragment listy obejmujący elementy od drugiego do trzeciego [2,3]
```


Indeksowanie po liście

```
lista_oryginal=[1,2,3,4,5]
lista_oryginal[:3] - elementy od pierwszego (domyślnie jeśli nie podano) do trzeciego: [1,2,3]
lista_oryginal[2:] - od trzeciego do ostatniego [3,4,5]
lista_oryginal[:] - wszystkie elementy listy [1,2,3,4,5]
lista_oryginal[-2:] - dwa ostatnie elementy [4,5]
lista_oryginal[:-1] - od pierwszego do przedostatniego [1,2,3,4]
lista_oryginal[3:-3] - od czwartego do czwartego od końca [3]
lista_oryginal[::-2] - Wyświetli wszystko, ale co drugi element listy [1,3,5]
lista_oryginal[::-1] - odwrócenie kolejności listy - wyświetlenie elementów od końca do początku [5, 4, 3, 2, 1]
```

Dodawanie elementu do listy

Zadanie:

Dodaj element o wartości 1000 i wstaw go na pozycję o indeksie 1. Pozostałe elementy w liście powinny zostać przesunięte

Zwróć uwagę na nietypowy zakres  `L=[1,2,3,4,5,6,7,8,9,10]`
`L[1:1]=[1000]`
`print(L)`
`[1, 1000, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Usuwanie elementu z listy

Źle:

```
L=[1,2,3,4,5,6,7,8,9,10]
L[0]=[]
print(L)
```

```
#[[], 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Dobrze:

```
L=[1,2,3,4,5,6,7,8,9,10]
L[0:1]=[]
print(L)
```

```
#[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Tą metodą można dodawać
usuwać i zamieniać elementy

Operacje na listach – zamiana elementów

Zadanie:

Zamień element 4 w liście na elementy 11, 22, 33, 44.

Tutaj trzeba podawać
zakres a nie jedną pozycję

```
L=[1,2,3,4,5,6,7,8,9,10]
L[3:4]=[11,22,33,44]
print(L)
```

```
[1, 2, 3, 11, 22, 33, 44, 5, 6, 7, 8, 9, 10]
```

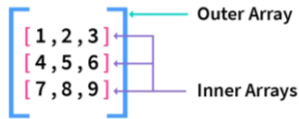
Zapisywanie list i tablic w programowaniu

$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}$ <p>Matrix</p>	=	$\begin{bmatrix} [1, 2, 3] \\ [4, 5, 6] \\ [7, 8, 9] \end{bmatrix}$ <p>2D Array</p>
---	---	---

$\begin{bmatrix} [1, 2, 3] \\ [4, 5, 6] \\ [7, 8, 9] \end{bmatrix}$	=	$\begin{bmatrix} [1, 2, 3] & [4, 5, 6] & [7, 8, 9] \end{bmatrix}$ <p>Memory</p>
---	---	---

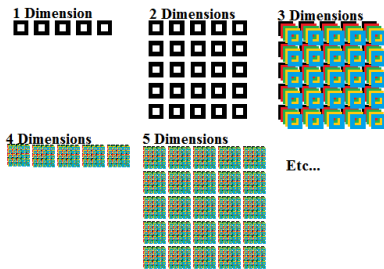
Źródło oraz więcej informacji na stronie: <https://www.scaler.com/topics/2d-array-in-python/>

Zapisywanie list i tablic w programowaniu



Źródło oraz więcej informacji na stronie: <https://www.scaler.com/topics/2d-array-in-python/>

Wymiary list i tablic w programowaniu



Czy na listach można wykonywać operacje matematyczne?

W Pythonie można wykonać wiele operacji matematycznych na listach.

1. Dodawanie list
2. Mnożenie listy przez liczbę całkowitą
3. Dzielenie elementów list
4. Odejmowanie elementów list
5. Pierwiastkowanie elementów list
6. Podnoszenie do potęgi elementów list
7. I inne...

Operacje matematyczne na listach

DODAWANIE

Dodawanie dwóch list to inaczej łączenie dwóch list w jedną całość
Aby dodać element do listy na stałe należy przypisać jej zawartość na nowo:

```
lista1=[1,2,3]
lista1 + ['nowy element'] # [1,2,3,'nowy element']
lista1 + [4, 5] # = [1,2,3,4,5]

lista2 = ['a','b','c']
lista1 + lista2 # = [1,2,3,'a','b','c']

# Przypisanie na nowo zawartości listy
moja_lista = moja_lista + ['dodaj element na stale']

#[1,2,3,'dodaj element na stale']
```

Operacje matematyczne na listach

ODEJMOWANIE

```
lista1 = [1, 2, 3, 4, 5]
lista2 = [3, 4, 5, 6, 7]

roznica_list = list(set(lista1) - set(lista2))

print(roznica_list)

# Wynik: [1, 2]
```

Czy my tu na pewno wykonujemy operacje na listach? Czy może na setach.

Operacje matematyczne na listach

MNOŻENIE Mnożenie listy przez liczbę całkowitą. Tego typu mnożenie nie jest wykonane na każdej liczbie z osobna.
Jest to tzw. powielanie listy.

```
lista = [1, 2, 3]
pomnozone_lista = lista * 3
print(pomnozone_lista)

# Wynik: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Operacje matematyczne na listach

DZIELENIE Można stosować operacje na elementach listy za pomocą list comprehension lub biblioteki numpy.

list comprehension

biblioteka numpy

```

lista1 = [1, 2, 3, 4, 5]
lista2 = [2, 2, 2, 2, 2]

iloraz_lista = [x/y for x,y in zip(lista1, lista2)]
print(iloraz_lista)
# Wynik: [0.5, 1.0, 1.5, 2.0, 2.5]

import numpy as np
lista1 = [1, 2, 3, 4, 5]
lista2 = [2, 2, 2, 2, 2]

iloraz_lista = np.array(lista1) /
np.array(lista2)
print(iloraz_lista)
# Wynik: array([0.5, 1. , 1.5, 2. , 2.5])

```

Zastępowanie elementu listy

Jeśli chcemy zmienić jakiś element listy, należy odwołać się do wybranego miejsca tej listy i podać jego nową wartość.

```

# Zastępowanie elementu listy
moja_lista[0] = 'zastap element'

```

W wyniku pierwszego zabiegu element listy o indeksie 0 zostanie zastąpiony przez nowy element i moja_lista będzie wyglądać tak: ['zastap element', 2, 3].

Operacje na listach

W tym przypadku zmienna p zawiera posortowaną kopię oryginalnej listy, a jednocześnie oryginalna lista listy pozostaje bez zmian.

```

lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

# Utwórz kopię listy przed sortowaniem
p = lista.copy()

# Sortowanie w miejscu
p.sort()

print("Oryginalna lista:", lista)
print("Posortowana lista w miejscu:", p)

```

Iterowanie po liście za pomocą pętli for i funkcji map

Metoda `map()` w języku Python jest funkcją wbudowaną, która pozwala na zastosowanie określonej funkcji do każdego elementu iterowalnego, takiego jak lista czy krotka.

```
# Przykładowa funkcja, która podnosi liczbę do kwadratu
def square(x):
    return x**2

# Przykładowa lista liczb
numbers = [1, 2, 3, 4, 5]

# Użycie map() do podniesienia każdej liczby do kwadratu
squared_numbers = map(square, numbers)

# Wynik map() jest obiektem map, więc przekształcamy go na listę dla czytelności
result_list = list(squared_numbers)

print(result_list)
#[1, 4, 9, 16, 25]
```

SŁOWNIKI

Słowniki to struktury podobne do list, jednakże nie pracują one w oparciu o indeksy a o parę klucza i wartości.

Nie posiadają one (w odróżnieniu od list) porządku (rozumianego przez kolejne indeksy).

Wszystkie klucze w słowniku muszą posiadać ten sam typ zmiennej, natomiast wartości mogą być różnych typów.

Odwoływanie się przez klucz słownika jest jednoznaczne, ponieważ w danym słowniku nie może być dwóch takich samych nazw kluczy.

Nazwy kluczy w słowniku są wrażliwe na wielkość liter!

Pamiętaj też, że przypisanie wartości do istniejącego już klucza automatycznie nadpisuje starą wartość.

```
# utworzenie słownika i wypisanie wartości przypisanej do Klucza2
słownik = {'Klucz1': 'String', 'Klucz2': 2023, 'Klucz3': 2.5, 'Klucz4': ['a', 'b', 'c']}
słownik['Klucz2']

#2023
```

SŁOWNIKI

W jaki sposób modyfikujemy elementy w słowniku?

```
# dodanie nowego elementu do słownika
słownik1 = {'Klucz1': 'C', 'Klucz2': 'Java'}
słownik1['Klucz3'] = 'Python'
słownik1

{'Klucz1': 'C', 'Klucz2': 'Java', 'Klucz3': 'Python'}
```

Przypisanie wartości do istniejącego już klucza automatycznie nadpisuje starą wartość.

```
# zmiana wartości elementu w słowniku
słownik1 = {'Klucz1': 'C', 'Klucz2': 'Java'}
słownik1['Klucz1'] = 'C#'
słownik1

Output:
{'Klucz1': 'C#', 'Klucz2': 'Java'}
```

SŁOWNIKI

Jak działa metoda pop() w słowniku?

```
#usunięcie elementu ze słownika ze zwróceniem wartości tego elementu
słownik1 = {'Klucz1':"C", 'Klucz2':"Java", 'Klucz3':"Python"}
słownik1.pop('Klucz2')

# 'Java'

#usunięcie elementu ze słownika
słownik1 = {'Klucz1':"C", 'Klucz2':"Java", 'Klucz3':"Python"}
del słownik1["Klucz3"]
słownik1

{'Klucz1': 'C', 'Klucz2': 'Java'}
```

Sprawdzanie zawartości słownika

```
#Utworzenie słownika
słownik3 = {'Klucz1':"R", 'Klucz2':"Java", 'Klucz3':"Python"}

#wyświetlenie kluczy w słowniku
print(słownik3.keys())

dict_keys(['Klucz1', 'Klucz2', 'Klucz3'])

#wyświetlenie wartości w słowniku
print(słownik3.values())

dict_values(['R', 'Java', 'Python'])

#wyświetlenie elementów
print(słownik3.items())

dict_items([('Klucz1', 'R'), ('Klucz2', 'Java'), ('Klucz3', 'Python')])
```

Słownik zagnieżdżony i łączenie słowników

```
#python umożliwia zagnieżdżanie słowników
słownik4 = {'klucz1':{'klucz11':{'klucz111': 'Python'}}}
słownik4['klucz1']['klucz11']['klucz111']

#Python,
# Podobny zapis znajduje się w pliku JSON

#modyfikacja dodanie do słownika 5 zawartości słownika 6
słownik5 = {'Klucz1':"JS", 'Klucz2':"Pascal"}
słownik6 = {'Klucz3':"Python"}
słownik5.update(słownik6)
słownik5

{'Klucz1': 'JS', 'Klucz2': 'Pascal', 'Klucz3': 'Python'}
```

SŁOWNIKI

Utworzenie słownika wygląda trochę inaczej niż w listach. Słowniki tworzy się jako parę „klucz – wartość” za pomocą nawiasów klamrowych {} i dwukropka :

```
dictionary1 = {'klucz1': 'wartosc1', 'klucz2': 'wartosc2'}
```

Słowniki charakteryzują się tym, że ich wartości mogą zawierać dowolny typ danych (np. łańcuchy, liczby, listy etc.). Z kluczami jest już inaczej- muszą być one zestawami tego samego typu elementów, np. napisy, liczby etc. Nie jest możliwe aby zestaw kluczy podać jednocześnie np. listę i liczby – Python zwróci wtedy błąd. Słownik z mieszanymi typami danych:

```
dictionary1 = {'key1': 'tekst', 'key2': ABC, 'key3': ['A1', 'A2', 'A3']}
# słownik zawierający różne typy danych
```

Wartości słownika wywołujemy przez odwołanie się do jego klucza:

```
dictionary1['key3']
# wywołanie klucza elementu słownika powoduje odniesienie się do jego wartości
```

Python zwróci: ['A1', 'A2', 'A3'].

Dodawanie i zamiana elementów w słowniku

Tworzymy słownik o nazwie dict1:

```
dict1={'key1': 'napis', 'key2': 123, 'key3': ['11', '12', '13']}
```

Dodawanie elementów do słownika dict1:

```
dict1['key4'] = [11, 22, 33]
# dodanie nowego elementu do słownika
```

Teraz słownik dict1 ma nowy element :

```
{'key1': 'napis', 'key2': 123, 'key3': ['11', '12', '13'], 'key4': [11, 22, 33]}
```

Wartości w słowniku można również modyfikować. Należy wtedy podać nazwę klucza, którego wartości chcemy zmienić i nową wartość.

```
dict1['key4'] = 'nowa wartość'
# modyfikowanie wartości słownika
```

Zmieniony słownik zawiera teraz inną wartość klucza 'key4':

```
{'key1': 'napis', 'key2': 123, 'key3': ['11', '12', '13'], 'key4': 'nowa wartość'}
```

Operacje na słownikach

Wartości słownika można również poddawać działaniom arytmetycznym. Można wykonywać różne operacje na wartościach przechowywanych w słownikach.

Wartości w słowniku można modyfikować poprzez:

dodawanie +,
odejmowanie -,
mnożenie/powielanie *,
dzielenie /.

#Dodawanie wartości za pomocą kluczy

```
dictionary = {'a': 1, 'b': 2, 'c': 3}
dictionary['d'] = dictionary['a'] + dictionary['b']
print(dictionary)
```

#wartość klucza a plus wartość klucza b jest równa wartości klucza d czyli 3

```
{'a': 1, 'b': 2, 'c': 3, 'd': 3}
```

Operacje na słownikach

Dodanie nowego łańcucha znaków do wartości klucza 1:

```
dict1 = {'key1': '1', 'b': 2, 'c': 3}

dict1['key1'] = dict1['key1'] + ' nowy'
print(dict1['key1'])
#1 nowy
```

```
dictionary = {'a': 5, 'b': 10, 'c': 15}

dictionary['a'] += 3
print(dictionary)

{'a': 8, 'b': 10, 'c': 15}
```

Operacje na słownikach

Mnożenie wartości elementów w słowniku:

```
dictionary = {'a': 2, 'b': 3, 'c': 4}
for key in dictionary:
    dictionary[key] *= 2
print(dictionary)

#{'a': 4, 'b': 6, 'c': 8}
```

Sumowanie wartości kluczy w słowniku:

```
dictionary = {'a': 5, 'b': 10, 'c': 15}
result = sum(dictionary.values())
print(result)

#30
```

Usuwanie elementu ze słownika

Wyróżnia się dwie metody usuwania elementów ze słownika.
Metodę del nazwa słownika[klucz]:

```
dictionary = {'key1': 5, 'key2': 10, 'key3': 15}
del dictionary['key2'] # usunięcie elementu za pomocą 'del'
print(dictionary)
#{'key1': 5, 'key3': 15}
```

Druga metoda usuwania elementów słownika to pop()

```
dictionary = {'key1': 5, 'key2': 10, 'key3': 15}
del dictionary['key2'] # usunięcie elementu za pomocą 'del'
dictionary.pop('key3')
print(dictionary)
#{'key1': 5}
```

Aby wyczyścić cały słownik używamy metody clear():

```
dictionary.clear()
#{}

Teraz słownik dictionary jest pusty.
```

ZBIORY

W języku Python zbiory (sets) są strukturami danych, które zawierają unikalne elementy, a ich porządek nie jest gwarantowany. Zwróć uwagę jak wyglądają nawiasy w listach a jak w zbiorach.

```
# Tworzenie zbioru z kilku elementów
set1 = {1, 2, 3, 4, 5}
print(set1)

# Tworzenie zbioru za pomocą konstruktora set(). Argumentem zbioru jest lista.
set2 = set([3, 4, 5, 6, 7])
print(set2)

{3, 4, 5, 6, 7}
```

Dodawanie i usuwanie elementu ze zbioru

```
# Dodawanie elementu do zbioru
set1.add(6)
print(set1)

# Usuwanie elementu ze zbioru
set1.remove(3)
print(set1)
```

Sumowanie zbiorów

Suma zbiorów to UNIA.

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

union_set = set1 | set2
print(union_set)
# Wynik: {1, 2, 3, 4, 5, 6, 7, 8}

union_set = set1.union(set2)
print(union_set)
# Wynik: {1, 2, 3, 4, 5, 6, 7, 8}
```

Część wspólna zbiorów

Przecięcie zbiorów to inaczej iloczyn zbiorów. W programowaniu nazywane jest intersekcją

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

intersection_set = set1 & set2
print(intersection_set)
# Wynik: {4, 5}

intersection_set = set1.intersection(set2)
print(intersection_set)
# Wynik: {4, 5}
```

Różnica zbiorów

Różnica zbiorów (Difference)

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

difference_set = set1 - set2
print(difference_set)
# Wynik: {1, 2, 3}

difference_set = set1.difference(set2)
print(difference_set)
# Wynik: {1, 2, 3}
```

Różnica symetryczna zbiorów

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

print(symmetric_difference_set)
# Wynik: {1, 2, 3, 6, 7, 8}

symmetric_difference_set =
set1.symmetric_difference(set2)
print(symmetric_difference_set)
# Wynik: {1, 2, 3, 6, 7, 8}
```

Wyższa Szkoła Przedsiębiorczości i Administracji



Wyższa Szkoła
Przedsiębiorczości
i Administracji

Programowanie

Funkcje w języku Python

Prowadzący: dr inż. Sylwester Korga

Własność materiałów edukacyjnych: dr inż. Sylwester Korga

Co to jest funkcja w językach programowania?

Funkcja w programowaniu to wydzielony fragment kodu, który można wielokrotnie używać w różnych miejscach programu.

Funkcja to konstrukcja programistyczna która może być już zawarta w języku programowania lub może być utworzona przez programistę.

Przykłady funkcji wbudowanych:

```
print()
input()
len()
max()
type()
str()
int()
sorted()
all()
enumerate()
filter()
```

Dlaczego w programowaniu korzystamy z funkcji?

Definiowanie funkcji w języku Python obejmuje utworzenie bloku kodu, który wykonuje konkretne zadania, a następnie przypisanie tej funkcji do określonej nazwy.

Oto ogólna składnia definicji funkcji w Pythonie:

```
def nazwa_funkcji(argument1,
argument2, ...):
    # Ciało funkcji
    # Wykonywane zadania
    return wynik
```

Terminy argument1, argument2, itd. to parametry, które funkcja przyjmuje. return jest opcjonalnym słowem kluczowym, które wskazuje, co funkcja powinna zwrócić. Oto przykładowa definicja i użycie funkcji w Pythonie:

Dlaczego w programowaniu korzystamy z funkcji?

Oto przykładowa definicja i użycie funkcji w Pythonie:

```
def dodaj(a, b):
    suma = a + b
    return suma

# wywołanie funkcji z argumentami
wynik = dodaj(3, 5)
print(wynik)
# program wydrukuje: 8
```

Zwróć uwagę, że nazwa funkcji jest przypisana do zmiennej.

Funkcje a automatyzacja kodu

Który z poniższych przykładów jest bardziej praktyczny - Wywołanie instrukcji po kolei za każdym razem kiedy ona jest potrzebna czy wywołanie funkcji?

```
def dodaj(a, b):
    instrukcja 1
    instrukcja 2
    instrukcja 3
    ...
    instrukcja n
    ...
    return x
```

Instrukcje w ciele pętli wykonywane są po kolei, jedna po drugiej.

Tworzony jest nowy obiekt <type 'function'> a referencję do niego umieszcza w zmiennej funkcja.

Z jakich elementów składa się funkcja w pythonie?

Definicja funkcji musi zawierać:

- 1) **nagłówek funkcji** obejmujący:
 - a) nazwę funkcji, która pozwoli zidentyfikować funkcję w pozostałej części programu
 - b) listę argumentów, która funkcja otrzymuje na początku działania programu
- 2) **ciało funkcji**, zawierające instrukcje, które zostaną wykonane w momencie wywołania (użycia) funkcji:
 - a) jeżeli funkcja ma zwracać jakiś rezultat, musi zawierać odpowiednią instrukcję

W języku Python składnia definicji funkcji jest następująca:

```
def dodaj(a, b):
    wynik=a+b
    return wynik
```

Jaka jest różnica pomiędzy deklaracją a definicją funkcji?

Deklarowanie funkcji (w języku C++) polega na wprowadzeniu informacji o funkcji do programu.

W tym momencie określamy nazwę funkcji, jej parametry (jeśli istnieją) oraz jej typ zwracany. Deklaracja funkcji informuje kompilator o istnieniu funkcji w programie, umożliwiając późniejsze odwołanie się do niej. Deklaracja funkcji nie zawiera jednak szczegółowej implementacji kodu w ciele funkcji.

Na przykład, deklaracja funkcji w języku C może wyglądać tak:

```
void funkcja(int paramet);
```

W przypadku języka Python, deklaracja funkcji jest równoważna z jej definicją, ponieważ Python jest językiem interpretowanym.

W Pythonie używamy słowa kluczowego "def" do zadeklarowania i zdefiniowania funkcji jednocześnie.

Natomiast definiowanie funkcji polega na dostarczeniu pełnej implementacji kodu w ciele funkcji. Definicja funkcji zawiera blok instrukcji, który zostanie wykonany, gdy funkcja będzie wywołana.

Co to znaczy zdefiniować funkcję?

Funkcje definiuje się używając słowa def. Po nim następuje nazwa *identyfikująca* funkcji, następnie para nawiasów, które mogą zawierać kilka nazw zmiennych jako argumentów. Linia kodu ze zdefiniowaną funkcją powinna być zakończona znakiem :

Przykład definiowania funkcji:

```
def hello(): # Zauważ, że ta funkcja nie posiada argumentów w nawiasie.
    # Blok instrukcji należący do funkcji.
    print('hello world')
# Koniec funkcji.
```

```
hello() # Wywołanie funkcji o nazwie hello.
```

```
hello() # Ponowne wywołanie funkcji o nazwie hello.
```

Co zostanie wyświetlone?

```
hello world
hello world
```

Gdzie powinny znajdować się funkcje?

W którym miejscu w programie powinny znajdować się funkcje?

W języku Python umieszczenie definicji funkcji w programie ma znaczenie z perspektywy organizacji kodu, czytelności i dostępności do funkcji w różnych miejscach programu.

Oto kilka ogólnych zaleceń dotyczących umieszczania funkcji w programie:

Na Początku Skryptu lub Modułu:

Čzęsto używa się konwencji, aby umieszczać definicje funkcji na początku skryptu lub modułu.

Ułatwia to zrozumienie struktury programu, ponieważ czytający kod może szybko zidentyfikować dostępne funkcje.

Gdzie powinny znajdować się funkcje?

Grupowanie Funkcji Według Zastosowania:

Jeśli funkcje wykonują podobne zadania lub są ze sobą powiązane tematycznie, można je grupować razem. Pomaga to w utrzymaniu porządku i zorganizowaniu kodu.

Przed ich Wywołaniem:

Jeśli funkcje są wywoływane w głównej części programu, to zazwyczaj umieszcza się ich definicje przed miejscem, w którym są używane. Zapewnia to, że interpreter Python wczytuje definicje funkcji przed ich użyciem.

W Plikach Nagłówkowych i Modułach:

W przypadku większych projektów, zazwyczaj umieszcza się definicje funkcji w oddzielnych plikach (modułach) lub plikach nagłówkowych. Umożliwia to podział kodu na logiczne jednostki i ułatwia jego utrzymanie.

Czy funkcja współpracuje ze zmiennymi w programie?

Uwaga: Funkcję dobrze jest przypisać do zmiennej. Przyjrzyj się poniższym przykładom:

Sposób prawidłowy przypisania funkcji do zmiennej:

`nazwa_zmiennej = nazwa_funkcji()`

Nowa nazwa staje się synonimem funkcji

Można ją teraz wywoływać zarówno przez `funkcja()` jak i przez `nazwa_zmiennej`

Sposób nieprawidłowy przypisania zmiennej do funkcji:

`funkcja = 1`

W ten sposób można na zawsze stracić możliwość jej wywołania

Co to są parametry funkcji?

Parametry funkcji i argumenty funkcji to dwa pojęcia związane z definicją i wywołaniem funkcji w języku programowania Python

Parametry funkcji:

Definicja: Parametry funkcji to zmienne, które są używane w definicji funkcji.

Miejsce: Parametry znajdują się w nawiasach okrągłych w nagłówku funkcji.

Przykład: W poniższym przykładzie `x` i `y` są parametrami funkcji dodaj:

```
def dodaj(x, y):
    wynik = x + y
    return wynik
```

Parametry są nazwami zmiennych, które funkcja używa do odbierania wartości od osoby wywołującej funkcję. Wartości przekazywane do tych parametrów podczas wywołania funkcji nazywane są argumentami funkcji.

Co to są argumenty funkcji?

Parametry funkcji i argumenty funkcji to dwa pojęcia związane z definicją i wywoływaniem funkcji w języku programowania Python

Argumenty funkcji:

Definicja: Argumenty funkcji to wartości przekazywane do funkcji podczas jej wywoływania.

Miejsce: Argumenty znajdują się w nawiasach okrągłych podczas wywoływania funkcji.

Przykład: W poniższym przykładzie 3 i 4 są argumentami funkcji dodaj

```
wynik = dodaj(3, 4)
```

Argumenty są rzeczywistymi wartościami, które przekazujesz do funkcji podczas jej wywoływania. Te wartości są przypisywane do odpowiednich parametrów funkcji.

Podsumowując, parametry to nazwy zmiennych używanych w definicji funkcji, podczas gdy argumenty to rzeczywiste wartości przekazywane do tych parametrów podczas wywoływania funkcji.

Argumenty ze słowem kluczowym i argumenty domyślne

W Pythonie istnieją argumenty funkcji ze słowem kluczowym (keyword arguments). Argumenty ze słowem kluczowym pozwalają na przekazywanie wartości do funkcji, używając nazw parametrów. Dzięki nim można przekazywać argumenty w dowolnej kolejności,

```
def opisz_osobe(imie, wiek, zawod):
    print(f"Imię: {imie}, Wiek: {wiek}, Zawód: {zawod}, Id: 5555")

# Przekazywanie argumentów ze słowem kluczowym
opisz_osobe(imie="Anna", wiek=25, zawod="Inżynier")
opisz_osobe(zawod="Programista", imie="Tomasz", wiek=30,)

#Imię: Anna, Wiek: 25, Zawód: Inżynier, Id: 5555
#Imię: Tomasz, Wiek: 30, Zawód: Programista, Id: 5555
```

W tym przykładzie używamy argumentów ze słowem kluczowym, przekazując wartości do funkcji poprzez podanie nazw parametrów (imie, wiek, zawod=). Kolejność argumentów nie ma znaczenia, ponieważ są one przypisywane do konkretnych parametrów na podstawie ich nazw.

Jak tworzyć zmienną wewnątrz funkcji?

W Pythonie zmienne, które są zadeklarowane wewnątrz funkcji, są zazwyczaj traktowane jako zmienne lokalne. Zmienne te są dostępne tylko wewnątrz zakresu funkcji i nie są widoczne poza nią.

Lokalność zmiennych: Zmienna zadeklarowana wewnątrz funkcji jest lokalna dla tej funkcji, co oznacza, że jej zakres życia ogranicza się do czasu trwania wykonania funkcji. Po zakończeniu funkcji, te zmienne przestają istnieć.

```
def przykladowa_funkcja():
    zmienna_lokalna = 10
    print(zmienna_lokalna)

przykladowa_funkcja()

nowa_zmienna = zmienna_lokalna + 1
```

Tę zmienną interpreter nie obsługuje, ponieważ została ona zadeklarowana wewnątrz funkcji. Jest to zmienna lokalna, która poza funkcją nie istnieje.

Czy można tworzyć takie same nazwy zmiennych w różnych funkcjach?

Unikalność nazw: Zmienne lokalne w różnych funkcjach mogą mieć te same nazwy, ponieważ są to osobne zakresy.

Takie same nazwy zmiennych
ale w różnych funkcjach.

```
def funkcja_a():
    zmienna_lokalna = 5
    print(zmienna_lokalna)

def funkcja_b():
    zmienna_lokalna = "Tekst"
    print(zmienna_lokalna)

funkcja_a() # Wydrukuje 5
funkcja_b() # Wydrukuje "Tekst"
```

Ten kod zadziałał ale należy
uniknąć takich powtórzeń

#5
#Tekst

Zakres lokalny zmiennej w funkcji

Czy zmienne posiadają zakres (obszar) w kodzie w którym są widoczne. Czy jeśli są wszędzie widoczne to mają taką samą wartość?

Każda zmienna ma swój zakres, czyli blok, w którym została zadeklarowana, zaczynając od miejsca zdefiniowania jej nazwy.

```
x = 50
print('początkowa wartość zmiennej x to ', x)
def funkcja1(x):
    print('wczytana wartość zmiennej x do funkcji to ', x)
    x = 2
    print('wartość zmiennej x wewnątrz funkcji zostaje zmieniona na nową wartość  
równą ', x)
funkcja1(x)
print('wartość zmiennej x poza funkcją', x)
# początkowa wartość zmiennej x to 50
# wczytana wartość zmiennej x do funkcji to 50
# wartość zmiennej x wewnątrz funkcji zostaje zmieniona na nową wartość równą 2
# wartość zmiennej x poza funkcją 50
```

Uwaga: tutaj zmieniliśmy wartość zmiennej tylko wewnątrz funkcji. A czy jest możliwość aby zmienić wartość zmiennej poza funkcją?

Do czego służy słowo kluczowe global?

Teraz funkcja func() nie ma zmiennej x w liście parametrów i modyfikuje globalną zmienną x przy użyciu słowa kluczowego global.

```
x = 50
print('początkowa wartość zmiennej x to ', x)

def func():
    global x
    print('wczytana wartość zmiennej x do funkcji to ', x)
    x = 2
    print('wartość zmiennej x wewnątrz funkcji zostaje zmieniona na nową wartość równą ', x)

func()
print('wartość zmiennej x poza funkcją', x)

#początkowa wartość zmiennej x to 50
#wczytana wartość zmiennej x do funkcji to 50
#wartość zmiennej x wewnątrz funkcji zostaje zmieniona na nową wartość równą 2
#wartość zmiennej x poza funkcją 2
```

Dostęp do zmiennych globalnych i lokalnych

Instrukcje zawarte **wewnątrz funkcji** mogą odczytywać wartości zmiennych utworzonych w blokach kodu zawierających definicję tej funkcji

O ile nie istnieją zmienne lokalne o takiej samej nazwie

Zmienne utworzone **na poziomie pliku** (poza jakąkolwiek funkcją) noszą nazwę zmiennych globalnych

Aby funkcja mogła zapisywać do tych zmiennych (zmieniać ich wartość), muszą być one jawnie zadeklarowane jako **global**

Jak podawać wynik- poprzez print() czy return?

Czy warto stosować return w funkcji?
Co to znaczy, że funkcja coś zwraca?

def add(a,b): return a+b	→	def add(a,b): print(a+b)
solution=add(4,6) print(solution)		solution=add(4,6) print(solution)
#10	→	#None

Wniosek: Wynik podajemy poprzez return

Czy funkcja może zwrócić wartości kilka razy?
Czy return może występować więcej niż jeden raz?

Co to znaczy że funkcja coś zwraca?

Wyrażenia return returuujemy do wyjścia z funkcji. Możemy opcjonalnie zwrócić w tym momencie jakąś wartość.

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Co zostanie wyświetlone?

3

Każda funkcja domyślnie otrzymuje na końcu return None, chyba że napiszesz własne return. Możesz to sprawdzić uruchamiając print(some_function()) gdzie funkcja some_function nie używa wyrażenia return W ten sposób:

```
def some_function():
    pass
```

Funkcja lambda- praktyczne zastosowanie:

Funkcja lambda używana jako argument funkcji map:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)

# Wynik: [1, 4, 9, 16, 25]
```

Funkcje lambda są przydatne, gdy potrzebujesz małego kawałka kodu do jednorazowego użycia i nie chcesz tworzyć pełnej funkcji za pomocą def. Warto jednak pamiętać, że funkcje lambda są ograniczone do jednego wyrażenia i są bardziej przeznaczone do prostych operacji. W przypadku bardziej złożonych funkcji zalecane jest korzystanie z pełnych definicji funkcji (def).

Lambda

Jaka jest różnica pomiędzy zastosowaniem print() a return?

Print jest używane do wyświetlania informacji na ekranie, podczas gdy return jest używane do zwracania wartości z funkcji. Często chcemy używać return, jeśli potrzebujemy wyniku funkcji do dalszego wykorzystania w programie

Co to są i do czego służą Dekoratory?

W języku Python dekorator to specjalna konstrukcja, która pozwala na modyfikację funkcji lub metody w sposób elastyczny i modularny. Dekoratory pozwalają dodawać funkcjonalność do funkcji bez bezpośredniej modyfikacji ich kodu. Oznaczane są symbolem @ przed definicją funkcji.

@dekorator ← Taki dekorator trzeba wyżej w kodzie opisać za pomocą funkcji zewnętrznej i wewnętrznej.
def funkcja():
 # kod funkcji

Dekoratory zmieniają również metody w klasach. Oto przykład:

```
class Klasa:
    @dekorator
    def metoda(self):
        # kod metody
```

@

Dekoratory to też funkcje? Tak!

Dekorator to zazwyczaj funkcja, która przyjmuje inną funkcję lub metodę jako argument i zwraca nową funkcję lub metodę, która zawiera dodatkową funkcjonalność.

```
def moj_dekurator(funkcja):
    def wewnetrzna_funkcja():
        print("Wykonano przed funkcją")
        funkcja()
        print("Wykonano po funkcji")
    return wewnetrzna_funkcja

@moj_dekurator
def przykladowa_funkcja():
    print("To jest przykladowa funkcja")

przykladowa_funkcja()
```

Opis dekoratora

Nazwa dekoratora ze znakiem @

Przykład zastosowania dekoratora

Zadanie:

Napisz dekorator dla funkcji dodaj tak aby oprócz obliczeń dodawane były informacje przed wynikiem i po wyniku.

```
def dekorator_do_dodawania(funkcja):
    def wewnetrzna_funkcja(a, b):
        print(f"Dodaję {a} i {b}.")
        wynik = funkcja(a, b)
        print(f"Wynik dodawania: {wynik}")
        return wynik
    return wewnetrzna_funkcja

@dekorator_do_dodawania
def dodaj(a, b):
    return a + b

# Przykład użycia:
wynik = dodaj(3, 5)
```

Funkcje w programowaniu obiektowym

W programowaniu obiektowym w Pythonie funkcje są często nazywane metodami, ponieważ są związane z konkretnymi obiektami lub klasami. Metody są funkcjami, które działają na danych obiektu i mogą mieć dostęp do jego stanu. W języku Python definiuje się metody wewnątrz definicji klasy. Oto przykład:

```
class KlasaPrzykladowa:
    def __init__(self, x):
        self.x = x
    def metoda1(self):
        print("To jest metoda 1")
    def metoda2(self, y):
        wynik = self.x + y
        return wynik

obiekt = KlasaPrzykladowa(5) #Uwaga na nietypowe przypisanie lewej strony do prawej. Tutaj używamy nawiasów by stworzyć instancję klasy jako x=10.
wynik = obiekt.metoda2(3) # Output: To jest metoda 1# Wywołanie metody i przypisanie wyniku do zmiennej
print(wynik) # Output: 8
```

Co to są funkcje magiczne?

Funkcje magiczne w Pythonie to specjalne metody, które zaczynają się i kończą podwójnym podkreśleniem (`__`). Te metody są nazywane również metodami specjalnymi lub metodami magicznymi. Są to funkcje, które mają specjalne znaczenie i są automatycznie wywoływane w określonych sytuacjach. Metody magiczne pozwalają na dostosowanie i kontrolę zachowań obiektów oraz umożliwiają programiście pracę na niższym poziomie z różnymi aspektami języka Python.

Funkcje magiczne umożliwiają dostosowywanie zachowań obiektów, a ich stosowanie jest powszechne w zaawansowanym programowaniu obiektowym w Pythonie. Często są nazywane też "dunder methods" (od słowa "double underscore") z powodu podwójnego podkreślenia w ich nazwach.

Podaj przykłady funkcji magicznych czyli metod specjalnych.

Przykłady funkcji magicznych to:

`__init__(self, ...)`: Konstruktor obiektu, wywołany podczas tworzenia instancji klasy.
`__str__(self)`: Zwraca reprezentację obiektu jako ciąg znaków, wykorzystywane przy wywoływaniu funkcji `str()` lub `print()`.
`__len__(self)`: Zwraca długość obiektu, używane przy wywoływaniu funkcji `len()`.
`__getitem__(self, key)`: Pozwala na indeksowanie obiektu, używane przy użyciu notacji nawiasów kwadratowych `[]`.
`__setitem__(self, key, value)`: Pozwala na przypisanie wartości do indeksu obiektu, używane przy użyciu notacji nawiasów kwadratowych `[]`.
`__call__(self, ...)`: Pozwala na wywołanie obiektu jak funkcji, używane, gdy obiekt jest wywoływany.

Co to są metoda specjalna `__init__`?

W języku Python, `__init__` to specjalna metoda, która jest wywoływana automatycznie podczas tworzenia nowej instancji klasy. Jest to często nazywane "konstruktorem" klasy. Służy do inicjalizacji atrybutów obiektu lub wykonania innych operacji, które mają miejsce tuż po utworzeniu instancji.

```
class MojaKlasa:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Tworzenie instancji klasy i automatyczne
# wywołanie __init__
obiekt = MojaKlasa(10, 20)

# Atrybuty x i y zostały zainicjowane
print(obiekt.x) # Wynik: 10
print(obiekt.y) # Wynik: 20
```

W tym przykładzie, po utworzeniu instancji `MojaKlasa` za pomocą `obiekt = MojaKlasa(10, 20)`, metoda `__init__` jest automatycznie wywoływana, a wartości 10 i 20 są przekazywane jako argumenty `x` i `y`. Wewnątrz metody `__init__`, te wartości są przypisane do atrybutów `x` i `y` obiektu.

__init__

Główne cele metody `__init__` to:

Inicjalizacja atrybutów: Przypisanie początkowych wartości atrybutom obiektu.

```
def __init__(self, x, y):
    self.x = x
    self.y = y
```

W tym przykładzie metoda1 i metoda2 mają dostęp do atrybutów `x` i `y` za pomocą składni `self.x` i `self.y`.

```
def metoda1(self):
    print(f"Metoda 1: x={self.x}, y={self.y}")
```

Dlatego wartości te przechodzą między różnymi metodami klasy, ponieważ są częścią instancji klasy.

```
def metoda2(self):
    print(f"Metoda 2: x={self.x}, y={self.y}")
```

Tworzenie instancji klasy

obiekt = MojaKlasa(10, 20)

Wywołanie metod

obiekt.metoda1() # Metoda 1: x=10, y=20

obiekt.metoda2() # Metoda 2: x=10, y=20

Łączenie metod specjalnych

Czy można metody specjalne ze sobą łączyć? Nawet trzeba!

```
class MojaKlasa:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'MojaKlasa_napis wywołany(x={self.x}, y={self.y})'

obiekt = MojaKlasa(10, 20)
print(obiekt)

#MojaKlasa_napis wywołany(x=10,y=20)
```